注重实战，通过9个典型应用案例剖析Visual C++网络编程技术

Visual C++

网络编程经典案例详解

超值赠品
免费奉送

梁 伟 等编著



18小时多媒体语音教学视频

另外赠送53小时相关教学视频、大量电子书

◎ 技术性强：讲解了目前最流行的网络通信技术

◎ 注重实战：穿插了75个实例和9个典型案例进行讲解

◎ 重点突出：剖析了Socket编程、多线程编程及同步和异步模式

◎ 案例典型：涵盖了Visual C++网络开发最典型和热门的应用

◎ 视频教学：专门配备了大量与内容配套的多媒体教学视频

清华大学出版社

Visual C++网络编程经典案例详解

梁伟　等编著

清华大学出版社
北　京

内 容 简 介

本书由浅入深、循序渐进地向读者介绍了 Visual C++网络编程的基础知识，并且在此基础上讲解了常见的 Visual C++网络编程技术及典型应用案例，最终使读者从根本上提高自身的编程水平，能够独立开发网络应用程序。本书内容包括网络编程基础知识、Socket 套接字编程基础、多线程技术、FTP 浏览器实例程序、网页浏览器实例程序、网络通信器、邮件收发器、实用播放器、网络文件传输器、P2P 网络播放器、Q 版聊天软件的实现、串口通信技术等。本书最后专门讲解了如何用 Visual C++实现发送手机短信的案例，其中具体讲解了串口通信编程的实现方法、所需要的硬件设备以及数据封装等知识。

本书配套光盘中提供了作者专门为本书录制的多媒体语音教学视频和本书所涉及的源代码，这些源代码都经过精心调试，在 Windows XP 和 Windows 2003 下测试通过。

本书适合广大用 Visual C++进行网络程序开发的人员和想进一步提升网络编程水平的人员阅读，尤其适合具有一定 C 语言基础和 C++语言基础的人员或大中专院校的学生阅读。

图书在版编目（CIP）数据

Visual C++网络编程经典案例详解 / 梁伟等编著. —北京：清华大学出版社，2010.5
ISBN 978-7-302-21972-9

Ⅰ. ①V… Ⅱ. ①梁… Ⅲ. ①C 语言－程序设计 Ⅳ. ①TP312

中国版本图书馆 CIP 数据核字（2010）第 018838 号

责任编辑：夏兆彦
责任校对：徐俊伟
责任印制：
出版发行：清华大学出版社　　**地　址**：北京清华大学学研大厦 A 座
http://www.tup.com.cn　　**邮　编**：100084
社 总 机：010-62770175　　**邮　购**：010-62786544
投稿与读者服务：010-62776969，c-service@tup.tsinghua.edu.cn
质量反馈：010-62772015，zhiliang@tup.tsinghua.edu.cn
印 刷 者：
装 订 者：
经　销：全国新华书店
开　本：185×260　**印　张**：29.5　**字　数**：734 千字
（附光盘 1 张）
版　次：2010 年 5 月第 1 版　**印　次**：2010 年 5 月第 1 次印刷
印　数：1～ 000
定　价：　元

产品编号：036227-01

前　言

随着计算机的广泛应用和网络的普及，人们的生活和工作与网络的联系越来越紧密。最初，各式各样的网站为大家提供各项服务。随着网络应用的深入，各种网络应用软件也层出不穷。从腾讯的 QQ，到迅雷下载工具，再到各种视频网站播放软件，每个应用软件都成为人们生活不可或缺的一部分。

由于微软操作平台 Windows 的广泛应用，Windows 网络技术受到越来越多的公司和技术人员青睐。为了方便大家更好地学习 Windows 网络编程技术，笔者花费一年时间来编写本书。在写作期间，征询很多网友的意见，几次易稿。本书删减大量篇幅的协议分析讲解，只保留最核心的部分。为了帮助大家更快进入开发，大幅扩充实际应用开发的内容。在此，对这些网友表示深深的感谢。

本书特色

1．由浅入深，循序渐进

为了方便读者学习，本书首先从 C/S 网络模型等网络编程基础知识开始向读者讲解。并在读者不断学习的过程中，引进新的知识点，鼓励读者独立修改各章中的实例程序。从而使读者可以边学习，边动手，更快地掌握 VC++网络编程知识。

2．按知识点进行讲解，理解深刻

由于 VC++的相关技术较多，因此很多读者都感觉无从下手。本书通过按照知识点进行讲解的方式，帮助读者解决这个问题。本书在第 1 篇中着重讲解了网络编程基础知识以及利用多线程实现异步套接字编程的方法，使读者具备网络编程相关的基础知识。在第 2 篇中，通过对各个网络实例程序的学习，读者可以对利用 Visual C++ 6.0 进行网络程序的开发过程以及各种方法有更深入的理解。

3．案例精讲，深入剖析

根据笔者的项目实践经验，不同的软件系统开发，其技术实现原理都是相似的，即一通百通。所以本书没有像其他书籍一样对同一个知识点进行重复讲解。本书选取最典型的实例——串口通信编程应用，向读者进行综合讲解。首先，在第 12 章中，对串口通信方面的基础知识进行详细讲解，使读者准确掌握串口通信的基础知识。然后，在第 13～14 章中，通过串口 API 函数以及 MFC 串口控件进行实例程序的编写，向读者讲解两种方法的优点。

4．配多媒体语音教学视频光盘

笔者专门为本书录制了大量的多媒体语音教学视频，以让读者更加直观地理解本书内容，提高学习效率。另外，本书的配书光盘中提供了本书涉及的实例源程序，以方便读者使用。

5．提供技术支持

为了方便读者学习，本书提供技术论坛 http://www.wanjuanchina.net。如果读者在学习中遇到什么问题，可以将这些问题发布在论坛中，我们会帮助大家及时解决这些问题。另外，读者也可以发邮件到 bookservice2008@163.com 获得技术支持。

本书内容

第 1 章：如果读者还是初学者，那么在本章中，读者将学习到什么是 OSI 七层模型等网络编程中常用的几种网络模型结构及其意义，并且还可以学习到 MFC 中相关的网络套接字类等。

第 2 章：详细讲解了网络套接字的寻址方式和字节传输顺序，并介绍了相关的 Socket 函数。根据网络传输协议的不同，并结合实例程序分别向读者介绍了 TCP 和 UDP 这两种常用的网络协议的编程流程。

第 3 章：主要向用户讲解多线程技术的基础知识以及实现方法，并以此为基础进一步向读者讲解实现异步套接字编程的方法及其技巧。

第 4 章：详细讲解了 FTP 浏览器的工作原理及其常用命令等相关知识，并向读者介绍了连接、登录 FTP 服务器的命令、编程方法。在本章最后，通过创建 FTP 的客户端实例程序，向读者综合讲解了 FTP 编程的编程技巧等。

第 5 章：着重向读者介绍了网页浏览器的工作原理以及 HTTP 请求和响应知识。通过制作浏览器的个性化界面，向读者介绍了工具栏的编程技巧以及实现方法。最后，在 VC++中通过创建实例程序，向读者讲解了编程步骤等。

第 6 章：向读者讲解了通信程序的通信原理。通过发送端和接收端程序的实例讲解，向读者分别介绍了通信双方的编程技巧以及套接字编程的实现方法。

第 7 章：首先向读者介绍了通过 API 函数直接调用 Windows 自带的邮件收发器程序，并详细讲解了 CreateProcess()函数的使用方法。在本章中，还向读者介绍了 SMTP 会话的整个过程。最后，综合前面所讲的知识实现了邮件收发器的实例程序。

第 8 章：向读者讲解了网络文件传输器的实现原理及其编程方法。通过编写服务器端和客户端程序分别向读者讲解双方的实现方法，并且对代码进行了详细的分析。

第 9 章：通过编写播放器实例程序，向用户讲解了在 MFC 中实现消息映射的方法，使读者对 MFC 框架程序的工作原理有了进一步的认识。并通过多线程通信和多媒体控制函数向读者讲解了播放器实例程序的编写方法。最后，还向读者着重讲解了如何实现播放器程序的搜索功能。

第 10 章：主要向读者介绍了当今流行的 P2P 流媒体技术及其网络模型。通过向程序界面中添加播放 MP3 文件的功能向读者介绍了如何进行界面的美化以及实现方法。在本章

中，着重向读者讲解进行 P2P 通信的双方如何实现数据传输和控制。

第 11 章：通过读者熟悉的 Q 版界面编程方法向读者介绍界面中各个控件的使用等。在本章中，封装了自定义的通信数据类 Cdata，并使用该自定义类进行本章实例程序的编写。

第 12 章：在本章中，着重向读者介绍串口通信的基础知识。通过本章的学习，读者对于串口通信编程会有一个比较清晰地知识结构，对于后面深入学习串口编程会起到很好的作用。

第 13 章：在本章中，结合第 12 章所介绍的串口基础知识，向读者分别讲解了如何使用 MFC 串口控件和串口 API 函数进行编程实现串口实例应用的方法和实现过程。其中，对实现过程中所使用的数据结构等进行了非常细致的讲解。

第 14 章：通过前面所有章节的知识进行综合，通过短信猫实现 VC++发送手机短信的实例程序。在本章中，主要是使读者感受开发综合项目，使其达到实战的效果。

适合阅读本书的读者

- 具备 C、C++等计算机语言基础的初学者。
- 具有一定编程经验的初、中级用户。
- 自学 VC++网络编程的大中专学生。
- 对于 VC++网络编程感兴趣的社会培训用户等。

本书作者及编委会成员

本书主要由梁伟编写。其他参与编写和资料整理的人员有陈世琼、陈欣、陈智敏、董加强、范礼、郭秋滟、郝红英、蒋春蕾、黎华、刘建准、刘霄、刘亚军、刘仲义、柳刚、罗永峰、马奎林、马味、欧阳昉、蒲军、齐凤莲、王海涛、魏来科、伍生全、谢平、徐学英、杨艳、余月、岳富军、张健和张娜。在此一并表示感谢。

本书编委会成员有欧振旭、陈杰、陈冠军、项宇峰、张帆、陈刚、程彩红、毛红娟、聂庆亮、王志娟、武文娟、颜盟盟、姚志娟、尹继平、张昆、张薛。

编者

目　　录

第 1 篇　Visual C++网络编程基础

第 2 篇 Visual C++网络编程典型应用

第 3 篇　Visual C++串口通信

第 1 篇　Visual C++网络编程基础

⏭ 第 1 章 Visual C++网络编程概述

⏭ 第 2 章 Socket 套接字编程

⏭ 第 3 章 多线程与异步套接字编程

第 1 章　Visual C++网络编程概述

Visual C++（后面简写为 VC）网络编程是指用户使用 MFC 类库（微软基础类库）在 VC 编译器中编写程序，以实现网络应用。用户通过 VC 编程实现的网络软件可以在网络中不同的计算机之间互传文件、图像等信息。本章将向用户介绍基于 Windows 操作系统的网络编程基础知识，其开发环境是 VC。在 VC 编译器中，使用 Windows Socket 进行网络程序开发是网络编程中非常重要的一部分。

1.1　网络基础知识

如果用户要进行 VC 网络编程，则必须首先了解计算机网络通信的基本框架和工作原理。在两台或多台计算机之间进行网络通信时，其通信的双方还必须遵循相同的通信原则和数据格式。本节将向用户介绍 OSI 七层网络模型、TCP/IP 协议以及 C/S 编程模型。

1.1.1　OSI 七层网络模型

OSI 网络模型是一个开放式系统互联的参考模型。通过这个参考模型，用户可以非常直观地了解网络通信的基本过程和原理。OSI 参考模型如图 1.1 所示。

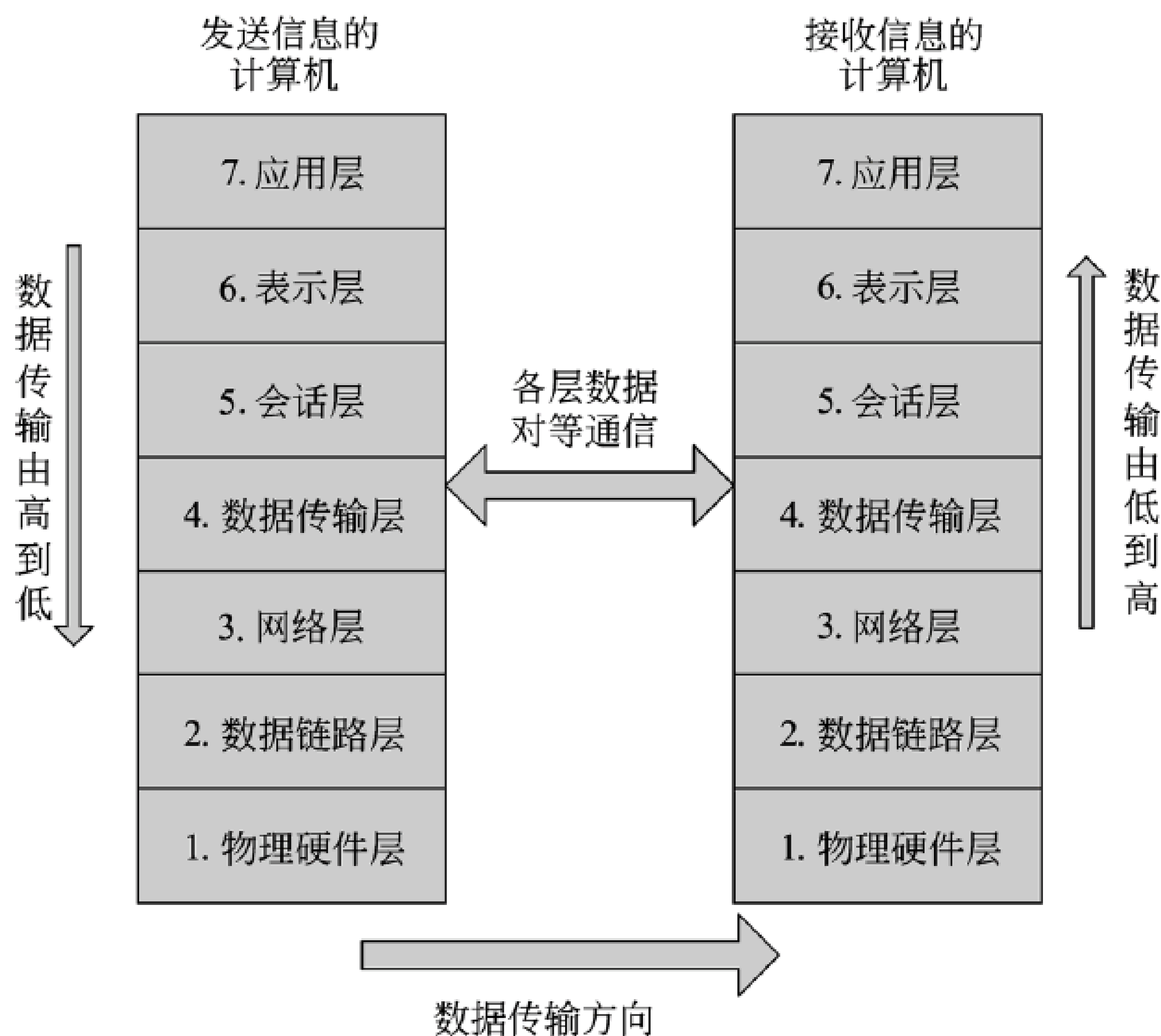


图 1.1　OSI 七层网络模型

用户从 OSI 网络模型可以很直观地看到，网络数据从发送方到达接收方的过程中，数据的流向以及经过的通信层和相应的通信协议。事实上在网络通信的发送端，其通信数据每到一个通信层，都会被该层协议在数据中添加一个包头数据。而在接收方恰好相反，数据通过每一层时都会被该层协议剥去相应的包头数据。用户也可以这样理解，即网络模型中的各层都是对等通信。在 OSI 七层网络模型中，各个网络层都具有各自的功能，如表 1.1 所示。

表 1.1　各网络层的功能

协 议 层 名	功 能 概 述
物理硬件层	表示计算机网络中的物理设备。常见的有计算机网卡等
数据链路层	将传输数据进行压缩与加压缩
网络层	将传输数据进行网络传输
数据传输层	进行信息的网络传输
会话层	建立物理网络的连接
表示层	将传输数据以某种格式进行表示
应用层	应用程序接口

注意： 在表 1.1 中列出了 OSI 七层网络模型中各层的基本功能概述。用户根据这些基本的功能概述会对该网络模型有一个比较全面的认识。

1.1.2　TCP/IP 协议

TCP/IP 协议实际上是一个协议簇，其包括了很多协议。例如，FTP（文本传输协议）、SMTP（邮件传输协议）等应用层协议。TCP/IP 协议的网络模型只有 4 层，包括数据链路层、网络层、数据传输层和应用层，如图 1.2 所示。

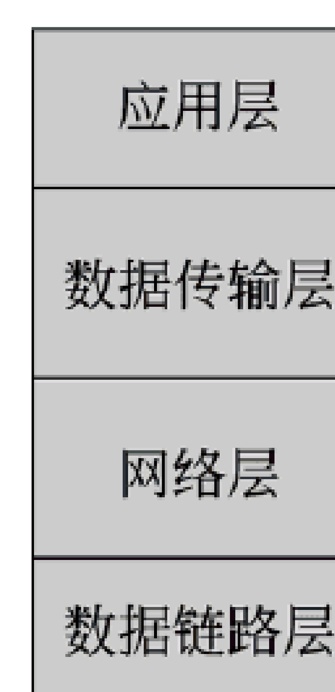


图 1.2　TCP/IP 网络协议模型

在 TCP/IP 网络编程模型中，各层的功能如表 1.2 所示。

表 1.2　TCP/IP网络协议各层功能

协 议 层 名	功 能 概 述
数据链路层	网卡等网络硬件设备以及驱动程序
网络层	IP 协议等互联协议
数据传输层	为应用程序提供通信方法，通常为 TCP、UDP 协议
应用层	负责处理应用程序的实际用于层协议

在数据传输层中，包括了 TCP 和 UDP 协议。其中，TCP 协议是基于面向连接的可靠的通信协议。其具有重发机制，即当数据被破坏或者丢失时，发送方将重发该数据。而 UDP 协议是基于用户数据报协议，属于不可靠连接通信的协议。例如，当用户使用 UDP 协议发送一条消息时，并不知道该消息是否已经到达接收方，或者在传输过程中数据已经丢失。但是在即时通信中，UDP 协议在对一些对时间要求较高的网络数据传输方面有着重要的作用。

1.1.3　C/S 编程模型

C/S 编程模型是基于可靠连接的通信模型。在通信的双方必须使用各自的 IP 地址以及端口进行通信。否则，通信过程将无法实现。通常情况下，当用户使用 C/S 模型进行通信时，其通信的任意一方称为客户端，则另一方称为服务器端。

服务器端等待客户端连接请求的到来，这个过程称为监听过程。通常，服务器监听功能是在特定的 IP 地址和端口上进行。然后，客户端向服务器发出连接请求，服务器响应该请求则连接成功。否则，客户端的连接请求失败。C/S 编程模型如图 1.3 所示。

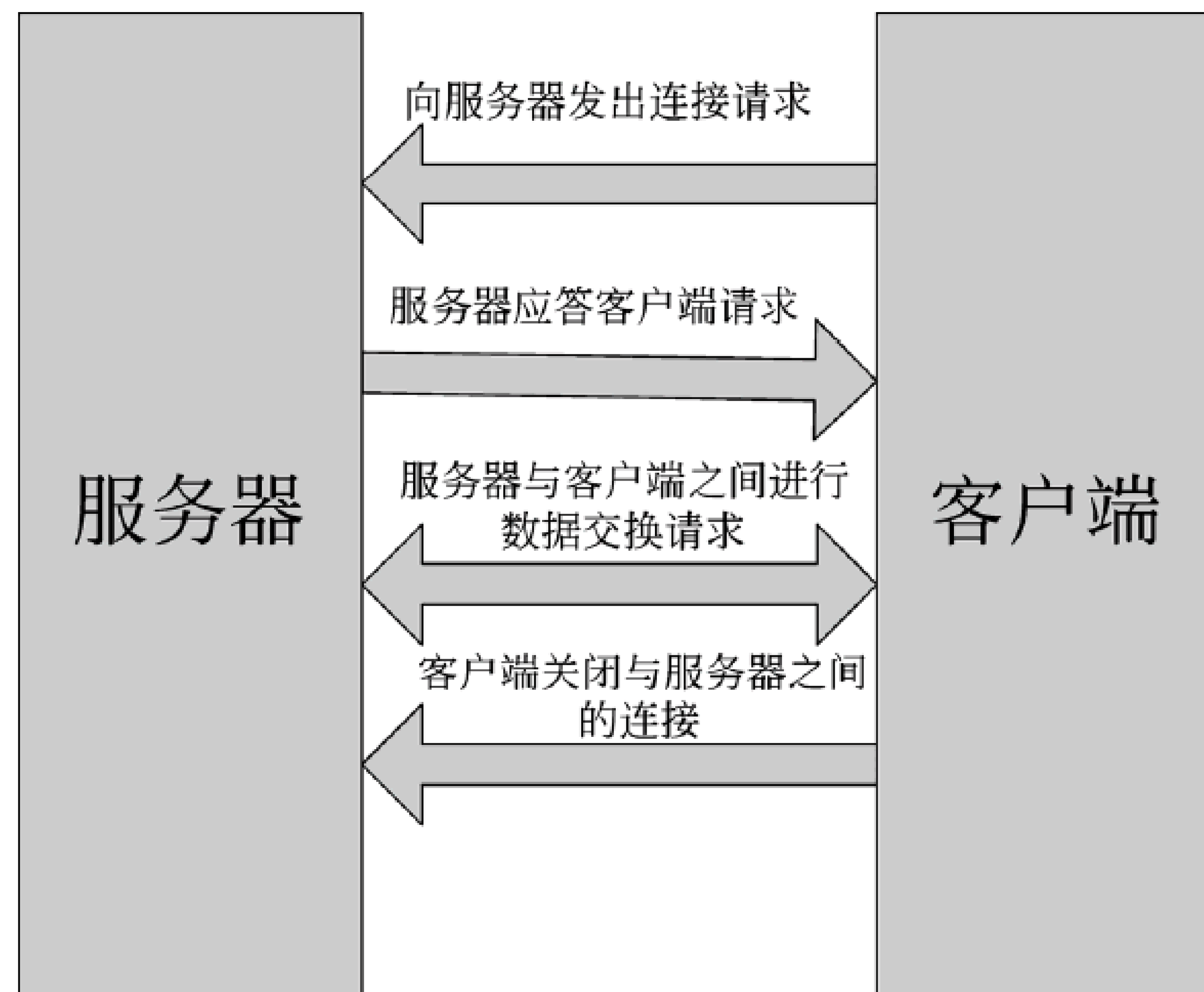


图 1.3　C/S 编程模型

由于客户端连接服务器时，需要使用服务器的 IP 地址和监听端口号才能完成连接。所以，服务器的 IP 地址和端口必须是固定的。在这里，向用户介绍部分协议所使用的端口号码。例如，HTTP 协议（网页浏览服务）所使用的端口号为 80，FTP 协议（文本传输）所使用的端口号是 21。

注意：用户在实际编程中，通信双方的连接以及数据通信均是基于 Socket（套接字）进行的。

1.2　网络编程基础

网络应用程序可以使用 MFC 中封装的套接字类进行编程，也可以使用 Windows API 函数进行程序开发。相比较而言，MFC 网络编程较简单一点，用户使用也非常方便。但是，使用 MFC 相关类编程会使用户对网络通信中的基本原理没有清晰的认识。而使用 Windows API 函数则恰好相反，可以使用户熟悉网络通信的基本原理。

1.2.1　Sockets 套接字

用户在 Windows 中编写网络通信程序时，需要使用 Windows Sockets（Windows 套接字）。与 Windows 套接字相关的 API 函数称为 Winsock 函数。

在网络通信的双方，均有各自的套接字，并且该套接字与特定的 IP 地址和端口号相关联。通常，套接字主要有两种类型，分别是流式套接字（SOCK_STREAM）和数据报套接字（SOCK_DGRAM）。其中，流式套接字是专门用于使用 TCP 协议通信的应用程序中，而数据报套接字则是专门用于使用 UDP 协议进行通信的应用程序中。

1.2.2　网络字节顺序

网络字节顺序是指 TCP/IP 协议中规定的数据传输使用格式，与之相对的字节顺序是主机字节顺序。网络字节顺序表示首先将数据中最重要的字节进行存储。例如，当数据 0x358457 使用网络字节顺序进行存储时，该值在内存中的存放顺序将是 0x35、0x84、0x57。因为通信数据可能会在不同的机器之间进行传输，所以通信数据必须以相同的格式进行整理。只有经过格式处理的通信数据，才能在不同的机器之间进行传输。

在 Winsock 中，已经提供了相关的函数处理网络字节顺序的相关问题，这些知识将在第 2 章中具体讲解。

1.3　Windows Sockets 介绍

在 MFC 类库中，几乎封装了 Windows Sockets 的全部功能。在本节中，将向用户介绍两个主要的套接字相关类，分别是 CAsyncSocket 类和 CSocket 类。

1.3.1　CAsyncSocket 类

在微软基础类库中，CAsyncSocket 类封装了异步套接字的基本功能。用户使用该类进行网络数据传输的步骤如下：

（1）调用构造函数创建套接字对象。

（2）如果创建服务器端套接字，则调用函数 Bind()绑定本地 IP 和端口，然后调用函数 Listen()监听客户端的请求。如果请求到来，则调用函数 Accept()响应该请求。如果创建客户端套接字，则直接调用函数 Connect()连接服务器即可。

（3）调用 Send()等功能函数进行数据传输与处理。

（4）关闭或销毁套接字对象。

注意： 在 MFC 中，所有类中均有一个变量 m_hWnd 表示该类的实例句柄。

1.3.2　CSocket 类

CSocket 类派生于 CAsyncSocket 类。该类不但具有 CAsyncSocket 类的基本功能，还

具有串行化功能。用户在实际编程中，通过将 CSocket 类与 CSocketFile 类和 CArchive 类一起使用，能够很好地管理数据以及发送数据。用户使用该类进行网络编程的步骤如下：

（1）创建 CSocket 类对象。

（2）如果创建服务器端套接字，则调用函数 Bind()绑定本地 IP 和端口，然后调用函数 Listen()监听客户端的请求。如果请求到来，则调用函数 Accept()响应该请求。如果创建客户端套接字，则直接调用函数 Connect()连接服务器即可。

（3）创建与 CSocket 类对象相关联的 CSocketFile 类对象。

（4）创建与 CSocketFile 类相关联的 CArchive 对象。

（5）使用 CArchive 类对象在客户端和服务器之间进行数据传输。

（6）关闭或销毁 CSocket 类、CSocketFile 类和 CArchive 类的 3 个对象。

1.4　小　　结

本章向用户介绍了网络编程有关的网络模型、工作原理、网络协议以及在 MFC 中使用相关的类进行网络程序编写步骤。用户通过本章的学习，将对网络编程的基础知识有一个大致的了解，同时也为后面的实际编程操作打下基础。如果用户在后面的编程实例中，遇到一些网络编程的基础知识疑问，可以再对本章进行复习、巩固，以便更好地理解网络编程知识。

第 2 章　Socket 套接字编程

套接字是由美国伯克利大学提出并设计的一种在网络中不同主机之间进行数据交换的通信桥梁。在实际生活中，人们所使用的网络通信软件功能均是基于 Socket 套接字作为通信桥梁实现。所以，套接字在网络编程中，有着非常重要的作用。本章将向用户介绍使用 Socket 套接字编程的相关概念以及实现方法。

2.1　寻址方式和字节顺序

在讲解套接字编程前，用户需要首先了解一下什么是寻址方式和字节顺序。在 Socket 套接字编程中，为了准确定位通信双方和数据传输的有效性、完整性，编程时必须使用统一的寻址方式和字节排列顺序。

2.1.1　寻址方式

因为套接字需要在各种网络协议中使用，所以为了区分程序所使用的网络协议必须使用统一的寻址方式。例如，在 TCP/IP 协议通信中，用户使用 IP 地址和端口号进行确定通信双方。而在其他的协议中不一定也使用该方式确定通信双方。

在 Winsock（Socket API）中，用户可以使用 TCP/IP 地址家族中统一的套接字地址结构解决 TCP/IP 寻址中可能出现的问题。该套接字地址结构定义如下：

```
struct sockaddr_in{
    short               sin_family;     //指定地址家族即地址格式
    unsigned short      sin_port;       //端口号码
    struct   in_addr    sin_addr;       //IP 地址
    char                sin_zero[8];    //留作备用，需要指定为 0
};
```

在这个结构中，成员 sin_family 指定使用该套接字地址的地址家族。在这里必须设置为 AF_INET，表示程序所使用的地址家族是 TCP/IP。

注意：该结构的最后一个成员并未实际使用，主要是为了与第一个版本的套接字地址结构大小相同而设置。在实际使用时，将这 8 个字节直接设为 0 即可。

该结构成员变量 sin_addr 表示 32 位的 IP 地址结构。其结构定义如下：

```
struct in_addr {
    union {
        struct{
```

```
            unsigned char s_b1, s_b2,s_b3,s_b4;
        } S_un_b;                          //用 4 个 u_char 字符描述 IP 地址
            struct {
            unsigned short s_w1,s_w2;
             } S_un_w;                     //用 2 个 u_short 类型描述 IP 地址

              unsigned long S_addr;        //用 1 个 u_long 类型描述 IP 地址
     } S_un;
};
```

通常，用户在网络编程中使用 1 个 u_long 类型的字符进行描述 IP 地址即可。例如，使用 IP 地址结构 in_addr 进行描述 IP 地址“218.6.132.5”。代码如下：

```
sockaddr_in addr;
addr.sin_addr.S_un.S_addr=inet_addr("218.6.132.5");
```

在程序中，首先定义 sockaddr_in 结构对象 addr，然后为 IP 地址结构 in_addr 中的成员 S_addr 赋值。因为结构成员 S_addr 所描述的 IP 地址均为网络字节顺序，所以程序调用 inet_addr()函数将字符串 IP 转换为以网络字节顺序排列的 IP 地址。

2.1.2　字节顺序

在 Socket 套接字编程中，传输数据的排列顺序以网络字节顺序和主机字节顺序为主。通常情况下，如果用户将数据通过网络发送时，需要将数据转换成以网络字节顺序排列，否则可能造成数据损坏。如果用户是将网络中接收到的数据存储在本地计算机上，那么需要将数据转换成以主机字节顺序排列。从数据存储的角度来讲，网络字节顺序即将数据中最重要的字节首先进行存储，而主机字节顺序则将不重要的字节首先存储。

注意：IP 地址结构 in_addr 中的成员 S_addr 的值均是以网络字节顺序排列。

1．字节顺序转换函数

在 Winsock 中提供了几个关于网络字节顺序与主机字节顺序之间的转换函数。函数定义如下：

```
u_short htons (u_short hostshort );
                    //将一个 u_short 类型的 IP 地址从主机字节顺序转换到网络字节顺序
u_long htonl (u_long hostlong );
                    //将一个 u_long 类型的 IP 地址从主机字节顺序转换到网络字节顺序
u_long ntohl (u_long netlong );
                    //将一个 u_long 类型的 IP 地址从网络字节顺序转换到主机字节顺序
u_short ntohs (u_short netshort );
                    //将一个 u_short 类型的 IP 地址从网络字节顺序转换到主机字节顺序
unsigned long inet_addr (const char FAR * cp);
                    //将一个字符串 IP 转换到以网络字节顺序排列的 IP 地址
char FAR * inet_ntoa (struct in_addr in);
                    //将一个以网络字节顺序排列的 IP 地址转换为一个字符串 IP
```

以上函数的使用均与操作系统平台无关。因此，用户使用这些函数编写的程序能在所有操作系统平台中运行。

2. 实例程序

在本节中，将编写实例程序向用户讲解字节顺序转换函数的用法。代码如下：

```
...                                     //省略部分代码
sockaddr_in addr;                       //定义套接字地址结构变量
in_addr in_add;                         //定义 IP 地址结构变量
addr.sin_family=AF_INET;                //指定地址家族为 TCP/IP
addr.sin_port=htons(80);                //指定端口号
addr.sin_addr.S_un.S_addr=inet_addr('127.0.0.1');
                                        //将字符串 IP 转换为网络字节顺序排列的 IP
char addres[]=inet_ntoa (addr.sin_addr.S_un.S_addr);
                                        //将网络字节顺序排列的 IP 转换为字符串 IP
```

在程序中，用户首先使用函数 inet_addr()将字符串 IP“127.0.0.1”转换为以网络字节顺序排列的 IP 并保存在 IP 地址结构成员 S_addr 中。然后，再使用函数 inet_ntoa()则将该成员所表示的 IP 值转换成字符串 IP。

2.1.3　Socket 相关函数

由于 Windows 网络程序开发均是基于 Windows 套接字实现，所以本节将重点介绍 MFC 中的 CSocket 类以及使用 CSocket 类编程的基本流程。

1. 创建套接字

使用 CSocket 类创建套接字对象是通过该类的构造函数创建的。其原型如下：

```
CSocket::CSocket( );
```

例如，用户创建 CSocket 类对象，代码如下：

```
CSocket sock;
```

如果用户需要创建套接字对象指针，则应该使用关键字 new 进行创建。代码如下：

```
CSocket *sock;                                  //定义套接字指针对象
sock=new CSocket;                               //使用 new 关键字创建套接字
```

2. 绑定地址信息

如果用户创建服务器套接字，那么用户应该调用该类的函数 Bind()将套接字对象与服务器地址信息绑定在一起。其原型如下：

```
BOOL Bind ( const SOCKADDR* lpSockAddr, int nSockAddrLen );
```

该函数的作用是将套接字对象与服务器地址结构绑定在一起。如果函数调用成功，则返回 true。否则，返回 false。参数 lpSockAddr 指定将要绑定的服务器地址结构，参数 nSockAddrLen 表示地址结构的长度。例如，用户将上面创建的套接字对象与地质结构绑定。代码如下：

```
CSocket sock;                                   //创建套接字对象
```

```
sockaddr_in addr;                                      //定义套接字地址结构变量
in_addr in_add;                                        //定义 IP 地址结构变量
addr.sin_family=AF_INET;                               //指定地址家族为 TCP/IP
addr.sin_port=htons(80);                               //指定端口号
addr.sin_addr.S_un.S_addr=inet_addr("127.0.0.1");      /
                                               /将字符串 IP 转换为网络字节顺序排列的 IP
sock.Bind((SOCKADDR*)addr,sizeof(addr));               //绑定套接字与地址结构
...                                                    //省略部分代码
```

在服务器端，当地址信息绑定套接字成功后，还需要调用函数 Listen()在指定端口监听客户端的连接请求。函数 Listen()的原型如下：

```
BOOL Listen( int nConnectionBacklog = 5 );
```

参数 nConnectionBacklog 表示套接字监听客户端请求的最大数目。该参数的有效范围是 1～5。默认为 5，表示该套接字只能监听 5 个客户端所发送的连接请求。例如，套接字监听 5 个客户端的连接请求，代码如下：

```
CSocket sock;                                          //创建套接字对象
sockaddr_in addr;                                      //定义套接字地址结构变量
in_addr in_add;                                        //定义 IP 地址结构变量
addr.sin_family=AF_INET;                               //指定地址家族为 TCP/IP
addr.sin_port=htons(80);                               //指定端口号
addr.sin_addr.S_un.S_addr=inet_addr("127.0.0.1");
                                           //将字符串 IP 转换为网络字节顺序排列的 IP
sock.Bind((SOCKADDR*)addr,sizeof(addr));               //绑定套接字与地址结构
sock.Listen(5);                                        //监听端口
```

3．连接服务器

客户端创建套接字成功以后，可以调用函数 Connect()向服务器发送连接请求。函数原型如下：

```
BOOL Connect( const SOCKADDR* lpSockAddr, int nSockAddrLen );
```

该函数调用成功，则返回 true。否则，将返回 false。参数 lpSockAddr 表示将连接的服务器地址结构。参数 nSockAddrLen 表示地址结构的长度大小。例如，服务器 IP 地址为“127.0.0.1”，端口为 80，客户端连接服务器，代码如下：

```
CSocket sock;                                          //创建套接字对象
sockaddr_in addr;                                      //定义套接字地址结构变量
in_addr in_add;                                        //定义 IP 地址结构变量
addr.sin_family=AF_INET;                               //指定地址家族为 TCP/IP
addr.sin_port=htons(80);                               //指定端口号
addr.sin_addr.S_un.S_addr=inet_addr("127.0.0.1");
                                           //将字符串 IP 转换为网络字节顺序排列的 IP
sock.Connect((SOCKADDR*)addr,sizeof(addr));            //连接服务器
```

4．数据交换

无论是服务器，还是客户端都是通过函数 Send()和 Receive()进行数据交换。函数原型如下：

```
virtual int Send( const void* lpBuf, int nBufLen, int nFlags = 0 );
virtual int Receive( void* lpBuf, int nBufLen, int nFlags = 0 );
```

其中，函数 Send()用于发送指定缓冲区的数据，函数 Receive()用于接收对方发送的数据，并将数据存放在指定缓冲区中。参数 lpBuf 表示数据缓冲区地址。参数 nBufLen 表示数据缓冲区的大小。参数 nFlags 表示数据发送或接收的标志，一般情况下，该参数均设置为 0。例如，使用这两个函数进行数据的发送和接收。代码如下：

```
...                                      //省略部分代码
char buff[]='a';                         //定义并初始化数据缓冲区
sock.Send(&buff,sizeof(buff),0);         //发送数据缓冲区中的数据
sock.Receive(&buff, sizeof(buff),0);     //接收数据并将数据存放在数据缓冲区中
```

5．关闭套接字对象

当服务器和客户端的通信完成以后，用户还必须调用函数 Close()将套接字对象关闭。否则，程序可能在退出时发生错误。该函数原型如下：

```
virtual void Close( );
```

例如，客户端关闭套接字对象，代码如下：

```
...                                      //省略部分代码
sock.Close();                            //关闭套接字对象
```

套接字关闭的同时，也将服务器与客户端之间连接关闭了。

本节主要向用户介绍了 CSocket 类的常用函数以及用法。当用户创建 VC 应用程序时，如果没有为应用程序指定支持 Windows Socket，那么用户必须手动添加该类的头文件 afxsock.h。否则，程序将不能使用 CSocket 类。

2.2　Winsock 网络程序开发流程

本节将向用户讲述基于 Windows Socket 的应用程序开发步骤，并将编写实例程序向用户介绍网络应用程序的开发过程以及 CSocket 类的具体使用方法。本节中的实例程序均在 VC 中进行编写、调试。

2.2.1　VC 中创建工程的步骤

用户在 VC 中使用应用程序向导创建基于套接字的应用程序工程时，必须为该应用程序指定支持 Windows Socket 功能。否则，创建的应用程序不能进行网络通信。

如果用户创建工程项目成功，则在应用程序向导设置的第二步，将询问用户是否需要在项目中支持 Windows Socket 功能，如图 2.1 所示。

如果用户在应用程序的第二步没有选择项目支持 Windows Socket 功能，则在程序中手动添加代码也可以达到同样的目的。其代码如下：

```
#include <afxsock.h>                     //包含 CSocket 类的头文件
```



图 2.1　支持 Windows Socket 功能

注意：头文件 afxsock.h 中包含了 CSocket 类的变量以及函数定义。

2.2.2　Winsock 编程流程

在本书的第 1 章中，已经向用户介绍了 Winsock 函数是用于网络编程的 Windows API 函数。本章在前一节中，向用户介绍了 CSocket 类的基本编程流程。所以，在本节中将向用户介绍使用 SOCKET API 函数进行网络程序开发的基本流程与方法。

1．初始化和释放套接字库

由于所有的 Winsock 函数均是从动态链接库 WS2_32.DLL 中导出的，但是，VC 在默认情况下并没有与该库进行连接。所以，用户需要在 VC 中进行相关设置，使其连接动态库 WS2_32.DLL。添加方法是选择“工程”|“设置”命令，将弹出 Project Settings 对话框，如图 2.2 所示。



图 2.2　添加动态链接库

用户在工程设置对话中，可以修改或添加库模块（如图 2.2 所示）。在库模块中添加动态链接库 WS2_32.DLL。这样，程序就可以调用 Winsock 函数了。

用户必须首先从动态链接库中调用函数 WSAstartup()对该库进行初始化，之后才能从该库中继续正确调用其他 Winsock 函数。否则，将出现错误。函数 WSAstartup()的原型如下：

```
int WSAStartup (WORD wVersionRequested,LPWSADATA lpWSAData);
```

该函数调用成功，将返回 0。否则，调用函数失败。参数 wVersionRequested 表示当前套接字库的版本号。例如，当前套接字版本号为 2.0，则将该参数设置为 2.0。代码如下：

```
WORD wVersionRequested=MAKEWORD(2,0);
```

参数 lpWSAData 指向结构体 WSADATA 的指针变量，表示获取到的套接字库详细信息。该结构体定义如下：

```
typedef struct WSAData {
        WORD wVersion;                              //库文件建议应用程序使用的版本号
        WORD wHighVersion;                          //库文件支持的最高版本
        char szDescription[WSADESCRIPTION_LEN+1];   //描述库文件的字符串
        char szSystemStatus[WSASYS_STATUS_LEN+1];   //系统状态字符串
        unsigned short iMaxSockets;                 //同时支持的最大套接字数
        unsigned short iMaxUdpDg;                   //已废弃
        char FAR * lpVendorInfo;                    //已废弃
} WSADATA, FAR * LPWSADATA;
```

用户初始化套接字库，代码如下：

```
WSAData data;                                       //定义 WSAData 变量
WORD wVersionRequested=MAKEWORD(2,0);               //定义套接字库版本号
::WSAStartup (wVersionRequested,&data);             //初始化套接字库
```

当程序退出时，用户还应该调用函数 WSACleanup()释放该套接字库。代码如下：

```
::WSACleanup();
```

2．创建套接字句柄

在 Socket API 中，创建套接字句柄的函数是 socket()。该函数原型如下：

```
SOCKET socket (
  int af,         //指定套接字所使用的地址格式，在本章中只能设置为 AF_INET
  int type,       //套接字类型
  int protocol    //如果参数 type 已经指定套接字类型为 TCP 或 UDP，则该参数可以设置为 0
);
```

该函数执行成功，将返回新创建的套接字句柄。否则，将返回 INVALID_SOCKET 表示失败。参数 type 的取值如表 2.1 所示。

表 2.1　套接字类型取值

套接字类型取值	含　义
SOCK_STREAM	创建流式套接字（基于 TCP 协议）
SOCK_DGRAM	创建数据报套接字（基于 UDP 协议）
SOCK_RAW	创建原始套接字（本书中未使用）

例如，创建流式套接字的句柄。代码如下：

```
SOCKET s;                                   //定义套接字句柄
s=::socket(AF_INET, SOCK_STREAM,0);         //创建并返回套接字句柄
```

3．绑定地址信息

对于服务器而言，套接字创建成功后，还应该将套接字与地址结构信息相关联。实现这一功能的函数是 bind()。该函数原型如下：

```
int bind (
  SOCKET s,                                 //套接字句柄
  const struct sockaddr FAR*  name,         //地址结构信息
  int namelen                               //地址结构的大小
);
```

该函数调用成功，则返回 0。否则，函数调用失败。例如，将套接字句柄绑定到本地地址，代码如下：

```
...                                         //省略部分代码
sockaddr_in addr;                           //定义套接字地址结构变量
in_addr in_add;                             //定义 IP 地址结构变量
addr.sin_family=AF_INET;                    //指定地址家族为 TCP/IP
addr.sin_port=htons(80);                    //指定端口号
addr.sin_addr.S_un.S_addr=INADDR_ANY        //表示服务器能够接收任何计算机发来的请求
::bind(s,(sockaddr)&addr,sizeof(addr));//绑定套接字到指定地址结构
```

当服务器程序将套接字句柄绑定套接字地址成功时，则调用函数 listen()实现监听端口的功能。该函数原型如下：

```
int listen (
  SOCKET s,                                 //实现监听功能的套接字句柄
  int backlog                               //指定监听的最大连接数量
);
```

该函数仅被用于流式套接字上。如果多个客户端同时向服务器发出连接请求，并且以及超过了最大监听数，则客户端将返回错误代码。例如，程序在已创建的套接字 s 上进行监听，代码如下：

```
...                               //省略部分代码
::listen(s,5);                    //在套接字上进行监听，并且将最大监听数指定为 5
```

4．连接

客户端程序连接服务器使用函数 connect()实现。函数原型如下：

```
int connect (
  SOCKET s,                                 //套接字句柄
  const struct sockaddr FAR*  name,         //将要连接的服务器地址信息结构指针
  int namelen                               //地址信息结构体长度
);
```

例如，客户端使用该函数连接地址为“127.0.0.1”，端口为 80 的服务器。代码如下：

```
sockaddr_in addr;                              //定义套接字地址结构变量
in_addr in_add;                                //定义 IP 地址结构变量
addr.sin_family=AF_INET;                       //指定地址家族为 TCP/IP
addr.sin_port=htons(80);                       //指定端口号
addr.sin_addr.S_un.S_addr=inet_addr("127.0.0.1");  //指定服务器地址
SOCKET s;                                      //定义套接字句柄
s=::socket(AF_INET,SOCK_STREAM,0);             //创建并返回套接字句柄
::connect(s,( sockaddr)&addr,sizeof(addr));    //连接服务器
...                                            //省略部分代码
```

如果服务器接收到客户端的连接请求，则可以调用函数 accept()接受该请求。函数原型如下：

```
SOCKET accept (
  SOCKET s,                                    //套接字句柄
  struct sockaddr FAR* addr,                   //获取连接对方的地址信息
  int FAR* addrlen                             //地址长度
);
```

该函数如果调用成功，则返回一个新的套接字句柄，用于通信双方数据的传输。

5．数据收发

当用户使用 Winsock 编程时，都是调用函数 send()和 recv()进行数据的发送和接收。函数原型如下：

```
int send (SOCKET s, const char FAR * buf, int len, int flags);
                                                    //发送数据函数
int recv (SOCKET s, char FAR* buf, int len, int flags); //接收数据函数
```

两个函数的各个参数以及表示的意义均相同。参数 buf 是指向数据缓冲区的指针变量，参数 flags 通常设置为 0。

注意：如果服务器使用上面的函数进行数据收发，则参数 s 应该为监听函数返回的新套接字句柄。如果客户端使用以上函数进行数据收发，则参数 s 应该为客户端创建的套接字句柄。

6．关闭套接字

当套接字使用完毕或程序退出时，用户应该调用函数 closesocket()关闭套接字句柄。函数原型如下：

```
int closesocket (
  SOCKET s                 //将关闭的套接字句柄
);
```

参数 s 表示即将关闭的套接字句柄。例如，用户关闭前面创建的套接字句柄 s，代码如下：

```
::closesocket(s);
```

本节主要向用户讲述了使用 Winsock 函数进行程序设计的基本流程，并讲解了部分常用函数的用法等知识。希望用户在实际编程的过程中，能不断地对本节知识进行回顾，加

深理解。

2.2.3　基于 TCP 的 Sockets 编程

在本节中，将编写一个简单的 TCP 服务器和 TCP 客户端程序。这两个实例程序均为控制台程序窗口。

1．TCP服务器

首先，在 VC 中新建一个基于控制台的应用程序工程，并将该工程命名为“TCP 服务器”，如图 2.3 所示。

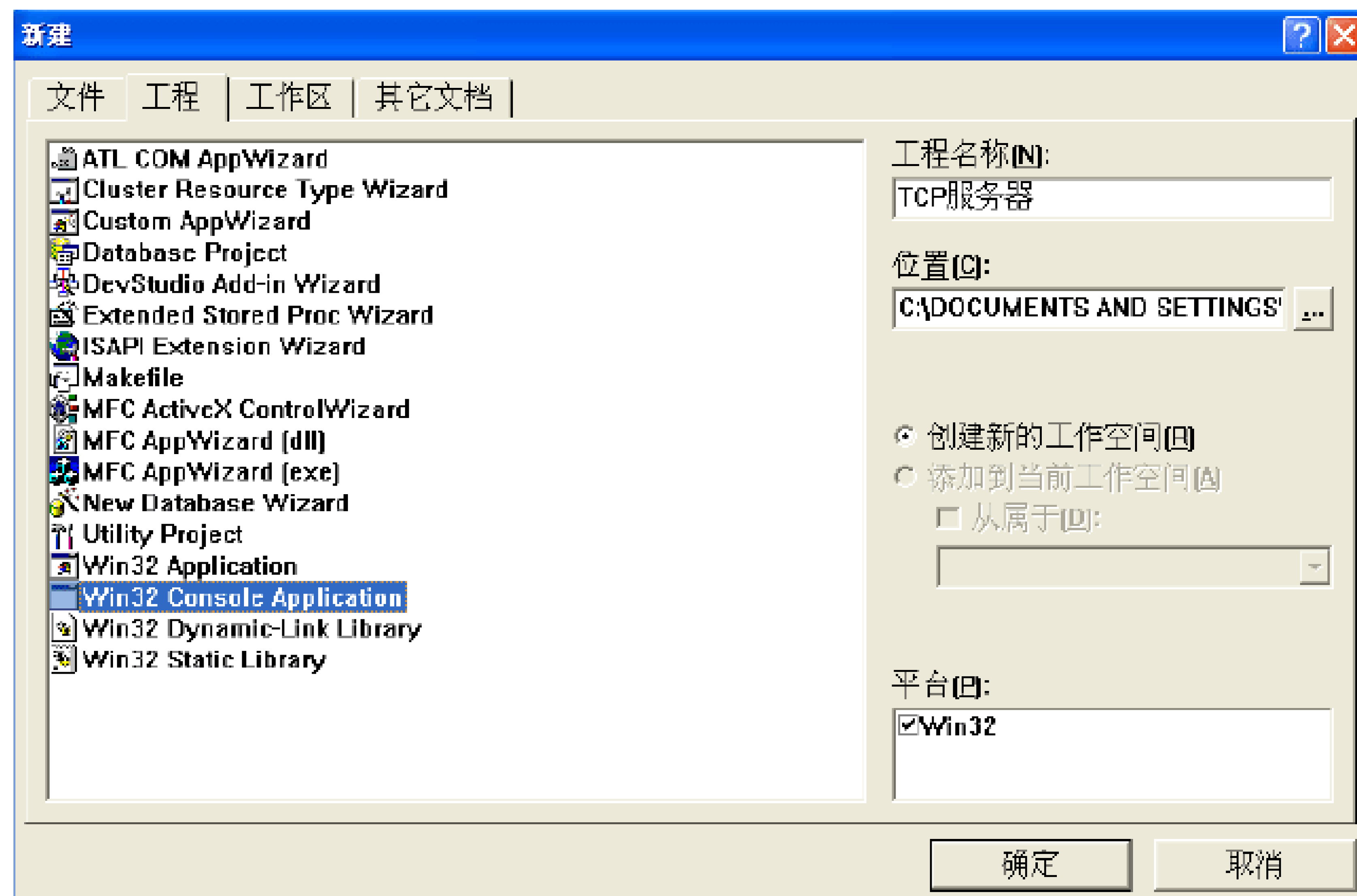


图 2.3　新建控制台应用程序

然后，单击“确定”按钮进行应用程序类型的设置。在本节中，将新建的控制台程序类型指定为一个空工程，如图 2.4 所示。

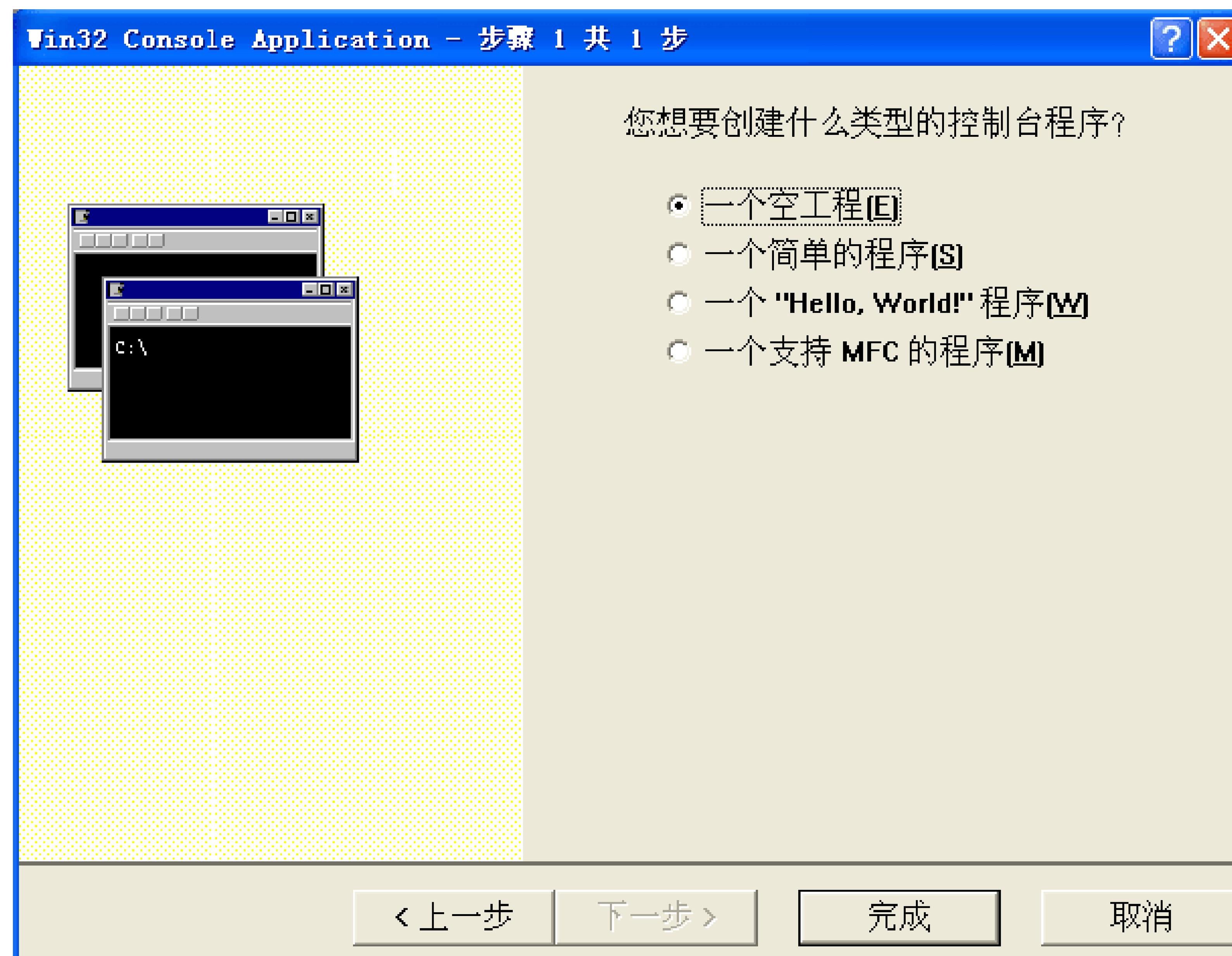


图 2.4　设置空的控制台程序

用户还需要在 VC 中添加一个空白的 C++源文件，名称为 TCPSEVER.cpp，如图 2.5 所示。

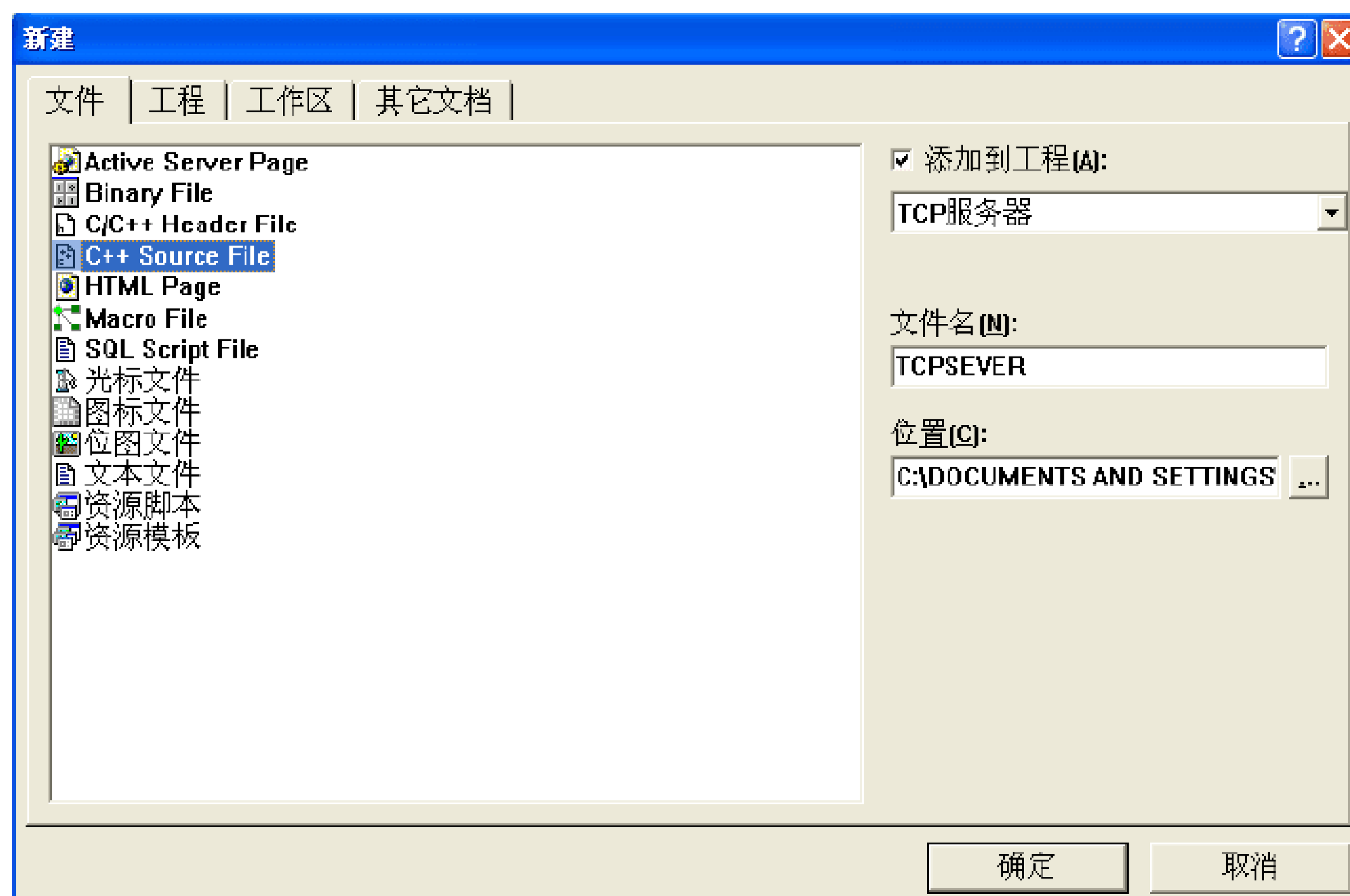


图 2.5　新建 C++资源文件

用户在新建的 C++源文件中进行代码编写。代码如下：

```
#include<winsock2.h>                        //包含头文件
#include<stdio.h>
#include<windows.h>
#pragma comment(lib,"WS2_32.lib")           //显式连接套接字库
int main()                                  //主函数开始
{
WSADATA data;                               //定义 WSADATA 结构体对象
WORD w=MAKEWORD(2,0);                       //定义版本号码
char sztext[]="欢迎你\r\n";                  //定义并初始化发送到客户端的字符数组
::WSAStartup(w,&data);                      //初始化套接字库
SOCKET s,s1;                                //定义连接套接字和数据收发套接字句柄
s=::socket(AF_INET,SOCK_STREAM,0);          //创建 TCP 套接字
sockaddr_in addr,addr2;                     //定义套接字地址结构
int n=sizeof(addr2);                        //获取套接字地址结构大小
addr.sin_family=AF_INET;                    //初始化地址结构
addr.sin_port=htons(75);
addr.sin_addr.S_un.S_addr=INADDR_ANY;
   ::bind(s,(sockaddr*)&addr,sizeof(addr));         //绑定套接字
::listen(s,5);                                      //监听套接字
printf("服务器已经启动\r\n");                         //输出提示信息
while(true)
{
    s1=::accept(s,(sockaddr*)&addr2,&n);            //接受连接请求
      if(s1!=NULL)
      {
          printf("%s 已经连接上\r\n",inet_ntoa(addr2.sin_addr));
          ::send(s1,sztext,sizeof(sztext),0);   //向客户端发送字符数组
            }
```

```
      ::closesocket(s);                           //关闭套接字句柄
      ::closesocket(s1);
      ::WSACleanup();                             //释放套接字库
      if(getchar())                               //如果有输入，则关闭程序
      {
      return 0;                                   //正常结束程序
        }
      else
      {
::Sleep(100);                                     //应用睡眠 0.1 秒
}   }  }
```

编译并运行程序，如图 2.6 所示。

服务器程序启动以后，如果没有客户端向其发送连接请求，则服务器将一直等待直到有客户端程序连接。

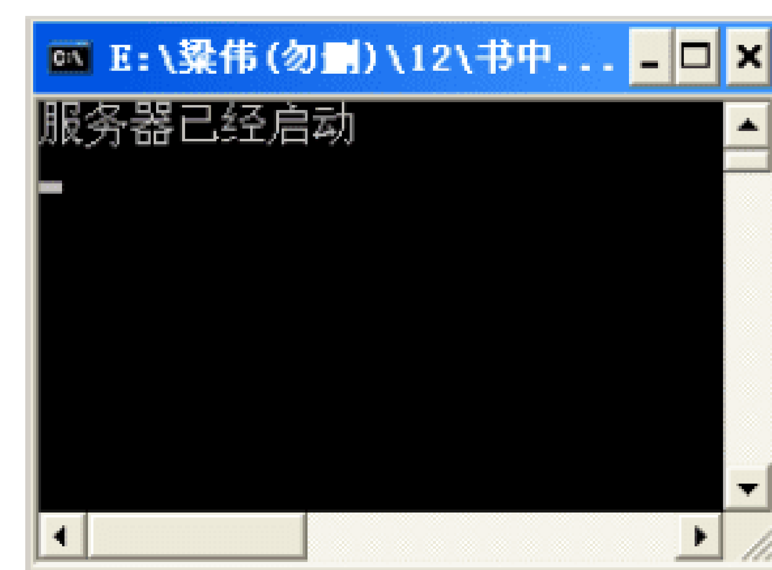


图 2.6　服务器启动界面

2. TCP客户端

在 VC 中创建基于控制台的应用程序，命名为“TCP 客户端”。其方法与 TCP 服务器的创建过程相同。所以，在这里不再赘述，请用户复习前面的相关内容。在新建的 C++源文件 TCPClient.cpp 中，用户可以编写客户端的功能代码。代码如下：

```
#include<winsock2.h>                          //包含头文件
#include<stdio.h>
#include<windows.h>
#pragma comment(lib,"WS2_32.lib")             //显式连接套接字库
int main()                                    //主函数开始
{
WSADATA data;                                 //定义 WSADATA 结构体对象
WORD w=MAKEWORD(2,0);                         //定义版本号码
::WSAStartup(w,&data);                        //初始化套接字库
SOCKET s;                                     //定义连接套接字和数据收发套接字句柄
char sztext[10]={0};
s=::socket(AF_INET,SOCK_STREAM,0);            //创建 TCP 套接字
sockaddr_in addr;                             //定义套接字地址结构
addr.sin_family=AF_INET;                      //初始化地址结构
addr.sin port=htons(75);
addr.sin addr.S un.S addr=inet addr("127.0.0.1");
printf("客户端已经启动\r\n");                     //输出提示信息
::connect(s,(sockaddr*)&addr,sizeof(addr));
::recv(s,sztext,sizeof(sztext),0);
printf("%s\r\n",sztext);
      ::closesocket(s);                       //关闭套接字句柄
      ::WSACleanup();                         //释放套接字库
      if(getchar())                           //如果有输入，则关闭程序
      {
      return 0;                               //正常结束程序
        }
      else
```

```
      {
::Sleep(100);                                    //程序睡眠
             }}
```

编译并运行程序，如图 2.7 所示。如果用户首先打开服务器程序，再打开客户端程序，则服务器会接受客户端的连接请求，而客户端会显示服务器发送的欢迎信息，如图 2.8 所示。

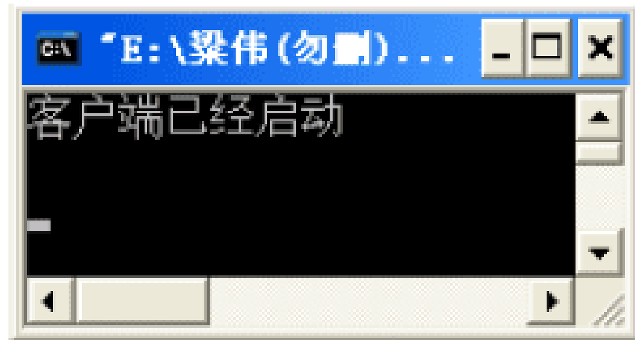


图 2.7　客户端启动界面

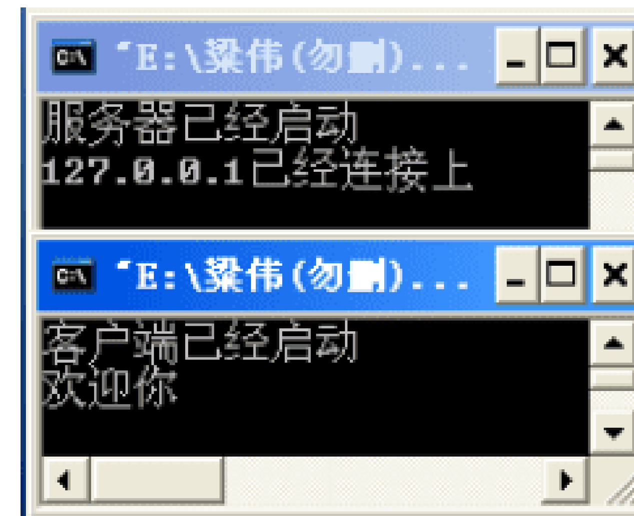


图 2.8　打开服务器与客户端

本节向用户讲解了 TCP 服务器与客户端的通信过程，并编写了实例代码。用户在学习的过程中，如果对本章实例有兴趣，可以将随书光盘中的相应的实例代码进行改写，以达到自己的要求。

2.2.4　基于 UDP 的 Sockets 编程

基于 UDP 的网络程序是面向无连接，不可靠的一种应用程序。所以，当程序创建套接字句柄成功以后，便可以直接调用函数进行数据收发，最后，关闭套接字对象。在整个过程中，程序都不用调用任何函数连接服务器或者接受客户端的连接等操作。这种类型的应用程序多用在即时通信中。

在 UDP 中进行数据收发的函数是 sendto()和 recvfrom()。函数原型如下：

```
int sendto (                           //发送函数
  SOCKET s,                            //套接字句柄
  const char FAR * buf,                //数据缓冲区
  int len,                             //数据的长度
  int flags,                           //一般设置为 0
  const struct sockaddr FAR * to,      //目标地址结构信息
  int tolen                            //目标地址结构大小
);
int recvfrom (SOCKET s, char FAR* buf, int len, int flags, struct sockaddr
FAR* from, int FAR* fromlen);          //接收函数
```

函数 recvfrom()的各个参数与函数 sendto()的参数基本一致。参数 from 是指向地址结构 sockaddr_in 的指针，表示数据发送方的地址信息。参数 fromlen 表示该地址结构的大小。

1．UDP服务器

首先，在 VC 中创建基于控制台程序窗口的应用程序，并命名为“UDP 服务器”，如图 2.9 所示。

然后，将该工程类型同样指定为空工程。在新建的工程中新建一个 C++源文件，名称为 UDPSever.cpp，如图 2.10 所示。

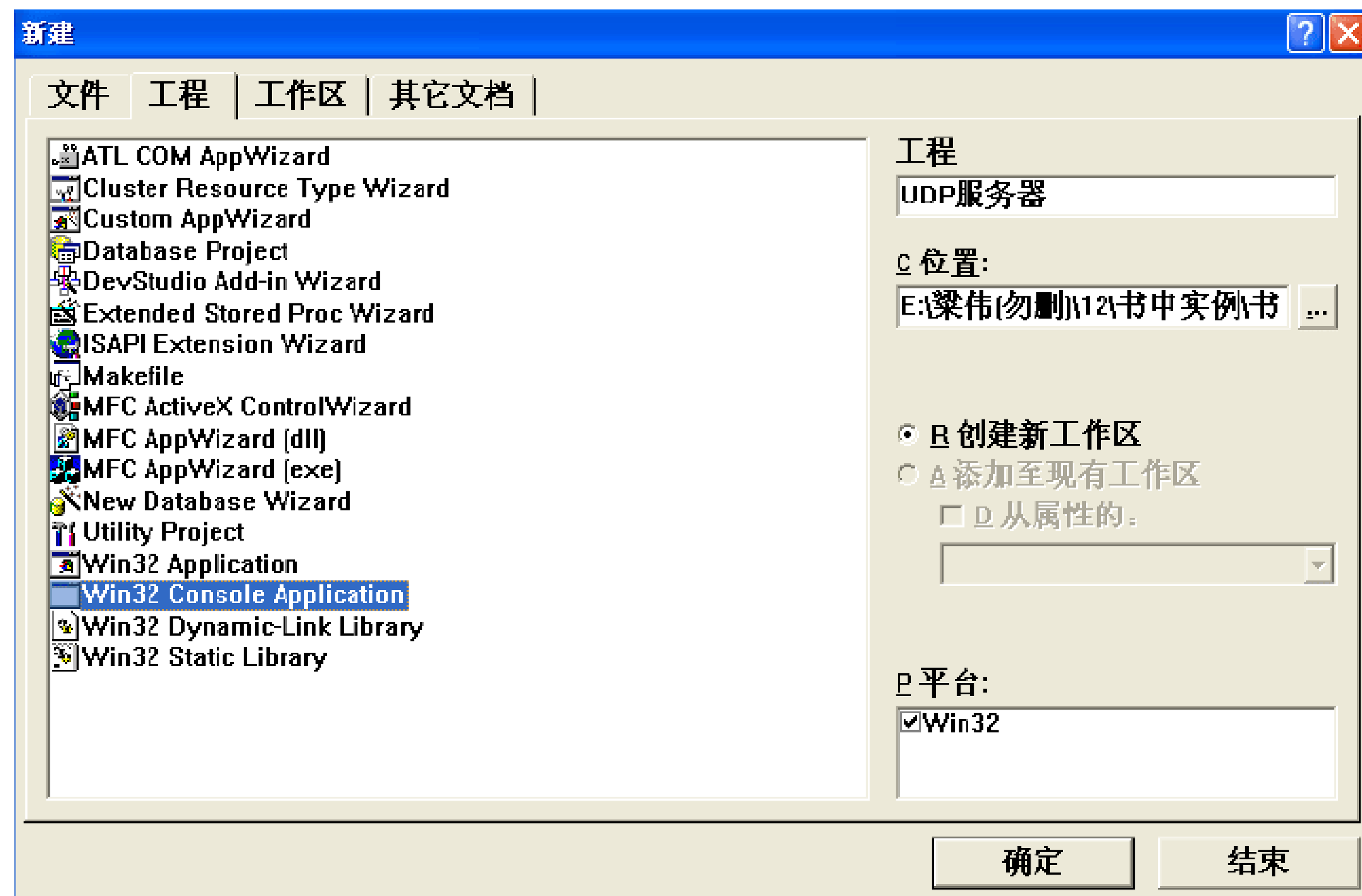


图 2.9　新建 UDP 服务器

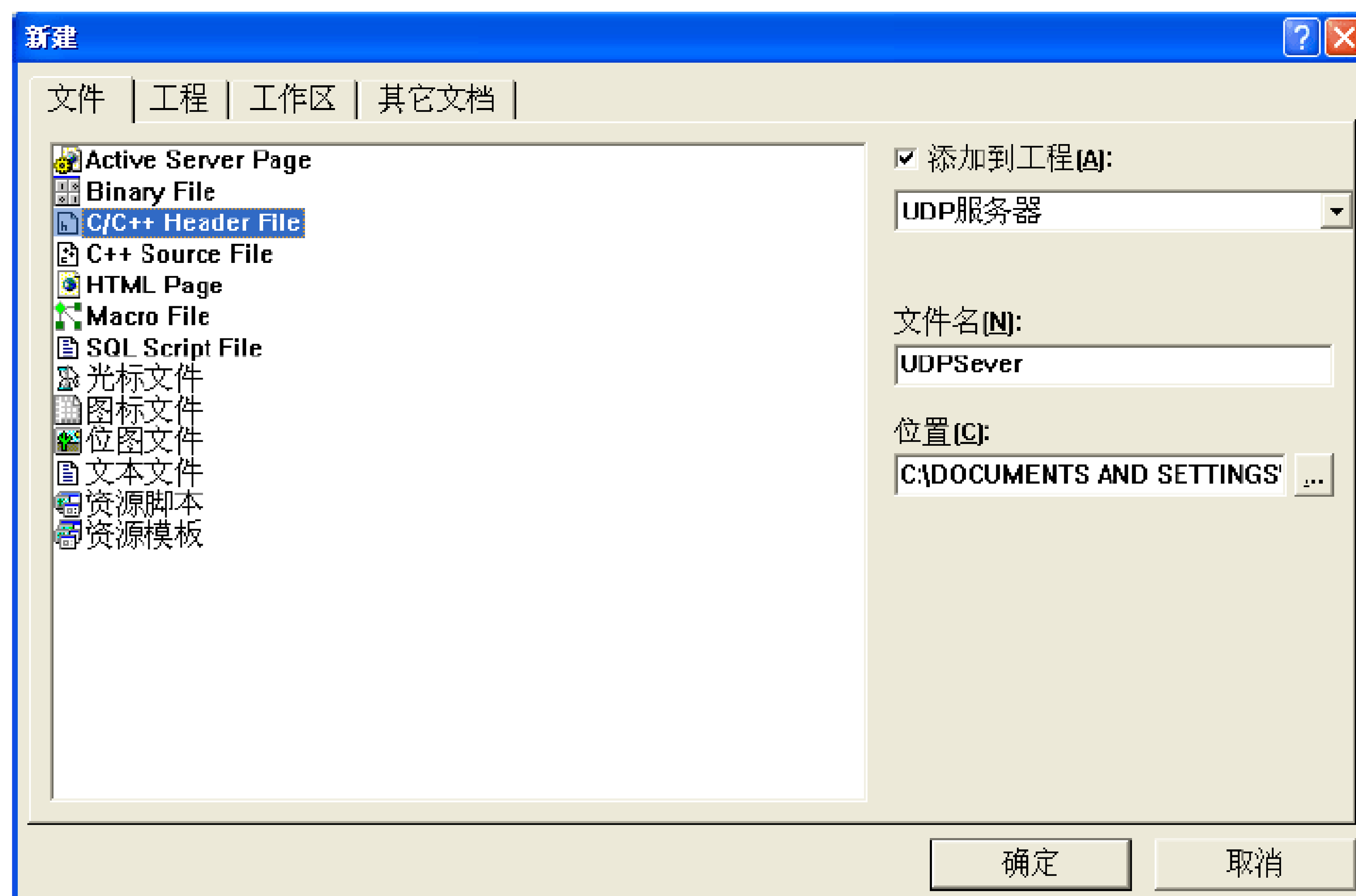


图 2.10　新建 C++源文件

现在用户可以在该源文件中编写 UDP 服务器的代码。代码如下：

```
#include<winsock2.h>                              //包含头文件
#include<stdio.h>
#include<windows.h>
#pragma comment(lib,"WS2_32.lib")                 //连接套接字库
int main()
{
WSADATA data;                                     //定义结构体变量
WORD w=MAKEWORD(2,0);                             //定义套接字版本
char sztext[]="欢迎你\r\n";                        //定义欢迎信息
::WSAStartup(w,&data);                            //初始化套接字库
```

```
SOCKET s;                                              //定义套接字句柄
s=::socket(AF_INET, SOCK_DGRAM,0);                     //创建 UDP 套接字
sockaddr_in addr,addr2;                                //套接字地址结构变量
int n=sizeof(addr2);                                   //地址结构变量大小
char buff[10]={0};                                     //接收数据缓冲区
addr.sin family=AF INET;
addr.sin_port=htons(75);
addr.sin_addr.S_un.S_addr=INADDR_ANY;
    ::bind(s,(sockaddr*)&addr,sizeof(addr));           //绑定套接字
printf("UDP 服务器已经启动\r\n");                       //显示提示信息
if(::recvfrom(s,buff,10,0,(sockaddr*)&addr2,&n)!=-1)         //接收客户端信息
      {
    printf("%s 已经连接上\r\n",inet_ntoa(addr2.sin_addr));
::sendto(s,sztext,sizeof(sztext),0,(sockaddr*)&addr2,n);//发送数据到客户端
    ::closesocket(s);                                  //关闭套接字对象
        ::WSACleanup();                                //释放套接字库
                 }
       if(getchar())                                   //如果有输入，则关闭程序
      {
      return 0;                                        //正常结束程序
        }
      else
      {
::Sleep(100);                                          //应用程序睡眠
          }
}
```

编译并运行程序，如图 2.11 所示。

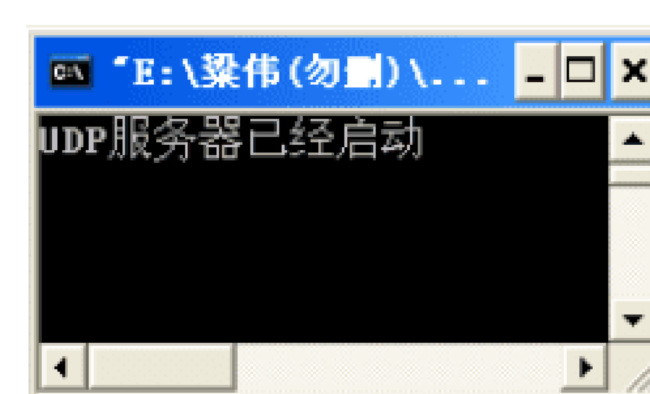


图 2.11　UDP 服务器启动界面

2．UDP客户端

在 VC 中创建 UDP 客户端程序时，与 UDP 服务器相同，工程类型均为空工程。所以，用户只需在 C++源文件中编写代码实现 UDP 客户端。代码如下：

```
#include<winsock2.h>                                      //包含头文件
#include<stdio.h>
#include<windows.h>
#pragma comment(lib,"WS2 32.lib")                         //连接套接字库
int main()
{
WSADATA data;                                             //定义结构体变量
WORD w=MAKEWORD(2,0);                                     //初始化套接字版本号
::WSAStartup(w,&data);                                    //初始化套接字库
SOCKET s;                                                 //定义套接字
s=::socket(AF INET,SOCK DGRAM,0);                         //创建 UDP 套接字
sockaddr in addr,addr2;                                   //定义套接字地址
int n=sizeof(addr2);
char buff[10]={0};
addr.sin family=AF INET;
addr.sin port=htons(75);
addr.sin addr.S un.S addr=inet addr("127.0.0.1");
printf("UDP 客户端已经启动\r\n");
if(::sendto(s,sztext,sizeof(sztext),0,(sockaddr*)&addr,n)!=0)  //发送信息
    {
    ::recvfrom(s,buff,10,0,(sockaddr*)&addr2,&n);  //接收信息
    printf("服务器说：%s\r\n",buff);
```

```
    ::closesocket(s);                                  //关闭套接字
        ::WSACleanup();                                //释放套接字库
                  }
      if(getchar())                                    //如果有输入，则关闭程序
      {
      return 0;                                        //正常结束程序
        }
      else
      {
::Sleep(100);                                          //应用程序睡眠
              }
}
```

编译并运行程序，如图 2.12 所示。

如果用户先启动 UDP 服务器，再启动 UDP 客户端，则会在服务器界面中显示客户端连接信息。而客户端界面中显示服务器发送的信息，如图 2.13 所示。

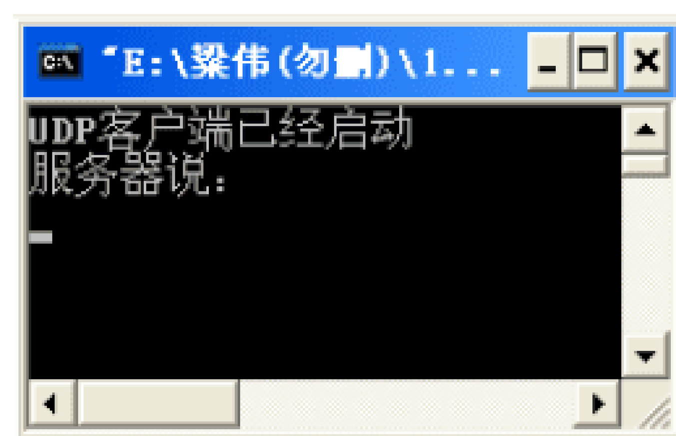


图 2.12 客户端启动界面

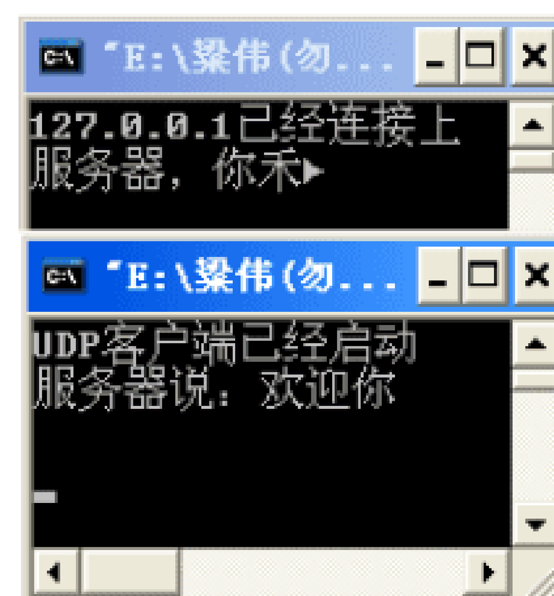


图 2.13 UDP 客户端与服务器进行通信

在本小节中，向用户讲解了在 VC 中使用 Winsock 函数进行网络程序开发，并结合 TCP 与 UDP 实例程序介绍了基于以上两种协议的网络程序编写方法。

2.3 网络程序实例应用

用户通过本章前面两节知识的学习，已经对网络程序的基本原理和程序编写方法有了进一步了解。在本节中，将引导用户在 VC 中编写基于对话框的 TCP 服务器和 TCP 客户端程序并且进行详细讲解。

2.3.1 TCP 客户端程序

在本小节中，将向用户介绍在 VC 中创建基于对话框模式的 TCP 客户端程序界面以及各个功能的实现等。

1. 创建工程

在 VC 中创建一个基于 MFC 的应用程序工程，并且将该工程名修改为“TCP 客户端程序”。步骤如下：

（1）选择“文件”|“新建”命令，打开“新建”对话框，如图 2.14 所示。

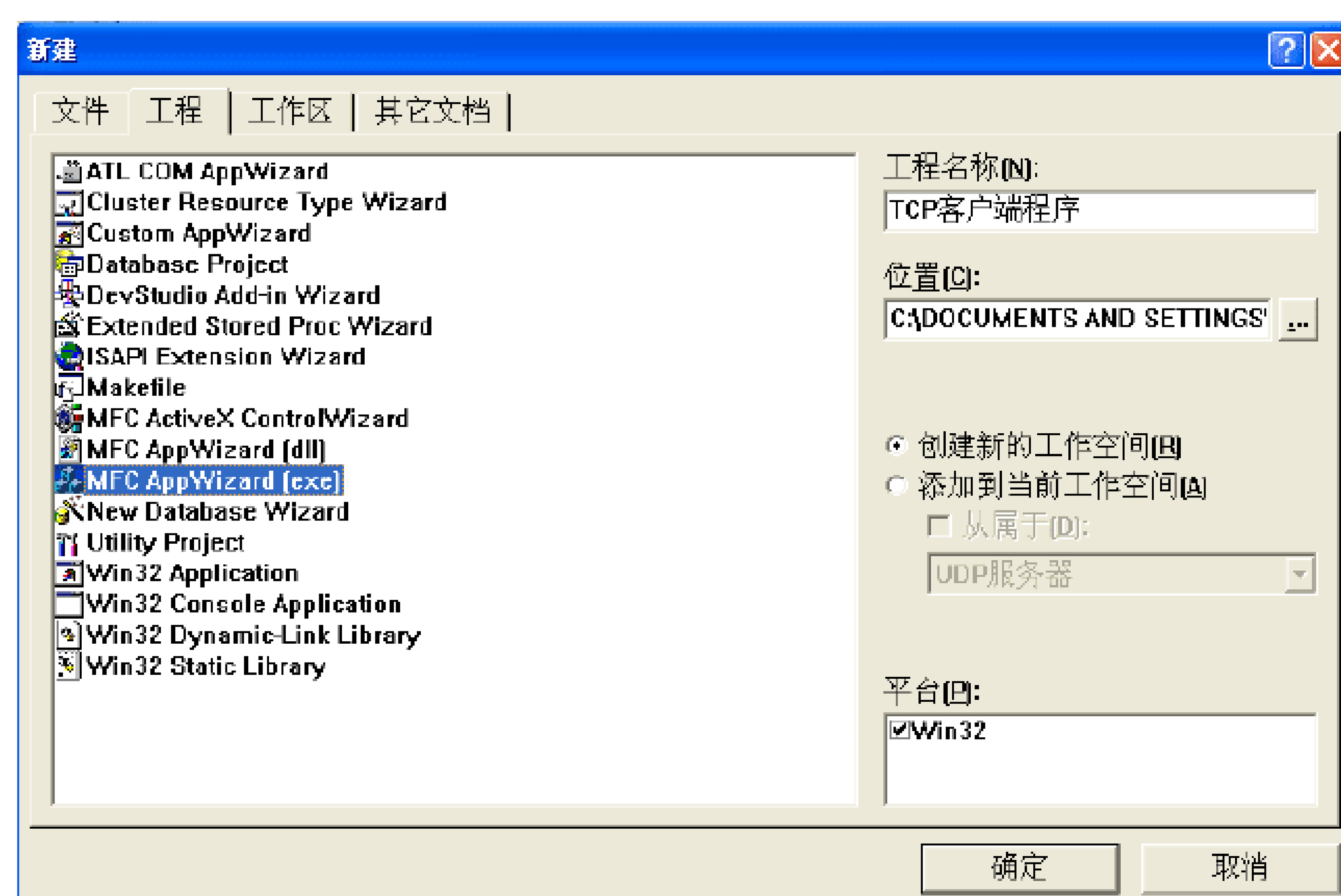


图 2.14　创建 TCP 客户端实例工程

（2）单击“确定”按钮，进入应用程序向导设置的第一步，修改应用程序的类型为“基本对话框”，如图 2.15 所示。

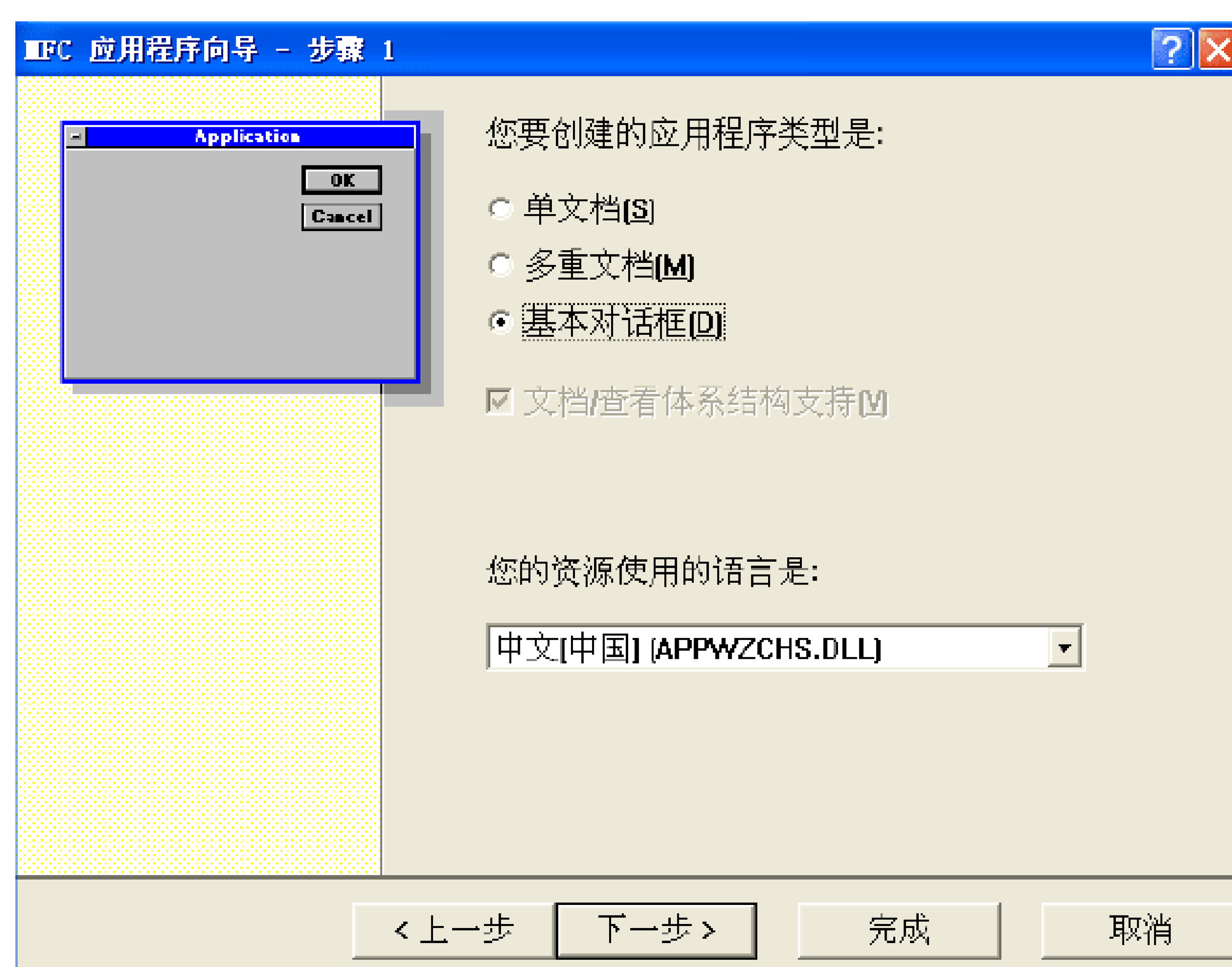


图 2.15　修改应用程序类型为基本对话框

（3）单击“下一步”按钮，进入应用程序向导设置的第二步，设置应用程序支持 Windows Socket 的功能，如图 2.16 所示。

（4）单击“完成”按钮，完成工程的创建以及相关配置。

现在，用户通过应用程序向导已经完成了 TCP 客户端工程的创建以及为该工程添加了支持 Windows 套接字功能等相关的一些配置。接下来，用户需要打开该工程的资源管理器进行程序界面的设计。

2．界面设计

当工程创建以后，用户可以打开资源管理器查看该工程的对话框资源，如图 2.17 所示。

用户可以通过向该对话框面板中添加相应的控件，以达到 TCP 客户端程序的基本功能，如图 2.18 所示。



图 2.16　设置应用程序支持套接字功能

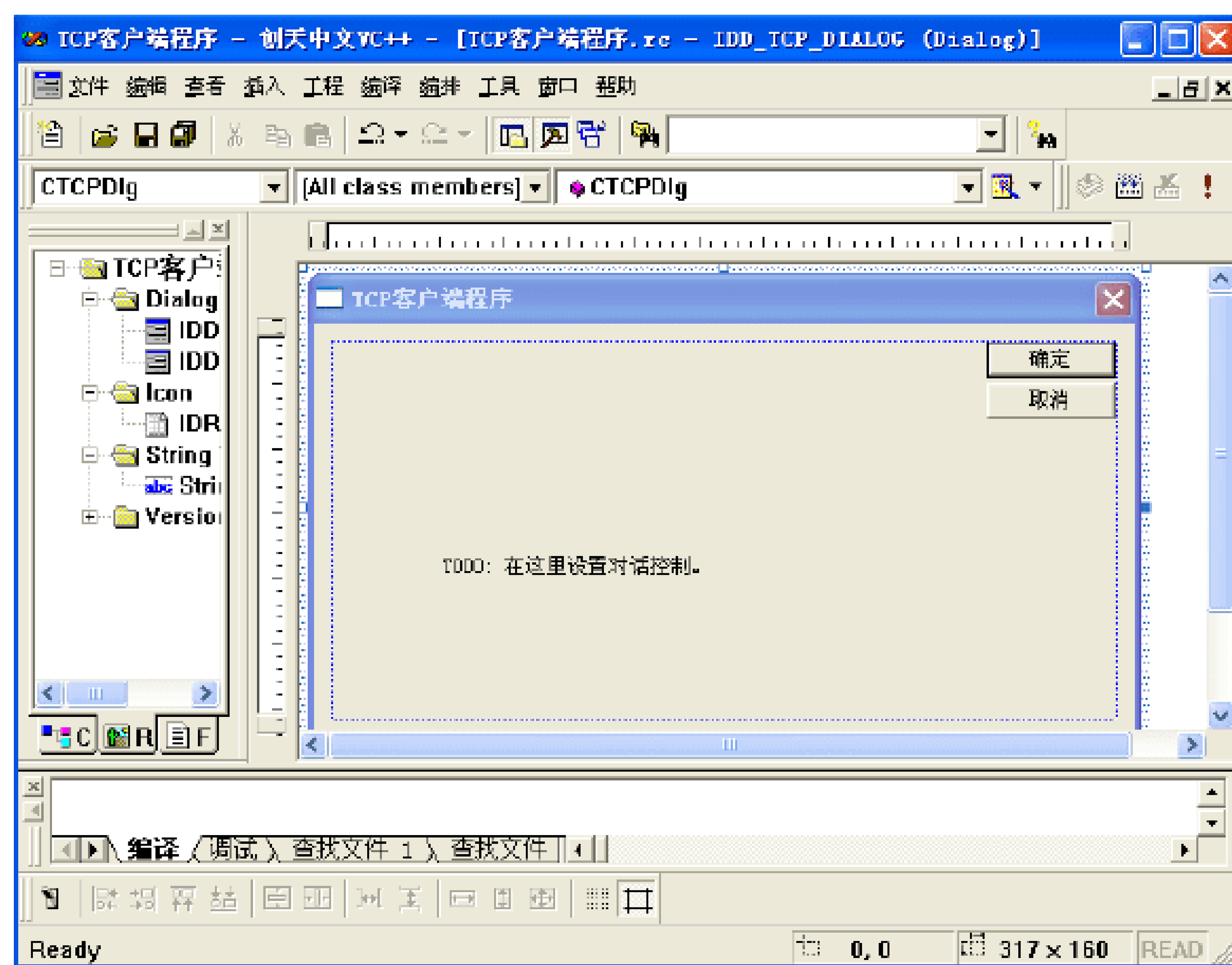




图 2.17　VC 默认情况下的对话框资源　　　图 2.18　完成设计后的界面效果

其中，用户添加了多个控件，新添加的控件 ID、类型以及作用，如表 2.2 所示。

表 2.2　控件ID、类型以及作用

控件 ID	控 件 类 型	控件在实例中的作用
IDC_ADDR	编辑框控件	输入服务器 IP 地址
IDC_PORT	编辑框控件	输入服务器端口
IDC_TEXT	编辑框控件	显示相关信息
IDC_SENDTEXT	编辑框控件	输入发送消息
IDC_SEND	按钮控件	发送消息按钮
IDC_CONNECT	按钮控件	连接服务器
IDC_STATIC1	静态控件	标识服务器地址
IDC_STATIC2	静态控件	标识服务器端口

3．界面初始化

TCP 客户端程序启动时，应该首先连接服务器以后，用户才能通过程序发送消息。所以，该程序初始化时的界面，如图 2.19 所示。



图 2.19　程序初始化界面

在界面初始化时，已经屏蔽了发送消息的功能。所以对于应用程序而言，避免了错误的发生。初始化代码如下：

```
BOOL CTCPDlg::OnInitDialog()
{
CDialog::OnInitDialog();
    ...                                                 //省略部分代码
GetDlgItem(IDC TEXT)->EnableWindow(false);              //禁用消息显示框
GetDlgItem(IDC SENDTEXT)->EnableWindow(false);          //禁用发送消息编辑框
GetDlgItem(IDC SEND)->EnableWindow(false);              //禁用发送消息按钮
return TRUE;
}
```

用户使用函数 GetDlgItem()获取对应 ID 控件的指针，然后使用该指针调用函数 EnableWindow()将控件禁用。函数 EnableWindow()的参数如果为 true，则表示该控件可以被使用。如果该参数为 false，则表示该控件被禁用。

在界面初始化时，除了初始化界面中的各按钮之外，还应该对套接字进行初始化。初始化套接字功能的代码应该在函数 CTCPDlg::OnInitDialog()中实现。代码如下：

```
class CTCPDlg : public CDialog                      //类声明
{
public:
 CTCPDlg(CWnd* pParent = NULL);
    ...                                             //省略部分代码
 SOCKET s;                                          //定义套接字对象
 sockaddr in addr;                                  //定义套接字地址结构变量
   }
BOOL CTCPDlg::OnInitDialog()
{
CDialog::OnInitDialog();
    ...                                             //省略部分代码
```

```
 s=::socket(AF_INET,SOCK_STREAM,0);              //创建套接字并返回其句柄
 return TRUE;
}
```

在代码中，用户在类中声明了套接字对象和套接字地址结构变量。然后，在初始化函数中创建了 TCP 套接字 s。

4．功能实现

在这一节中，用户可以为各个功能控件编写相应的代码，以实现其功能。首先，为“连接”按钮添加消息响应函数。在该控件上双击鼠标，将弹出 Add Member Function（添加成员函数）对话框，如图 2.20 所示。

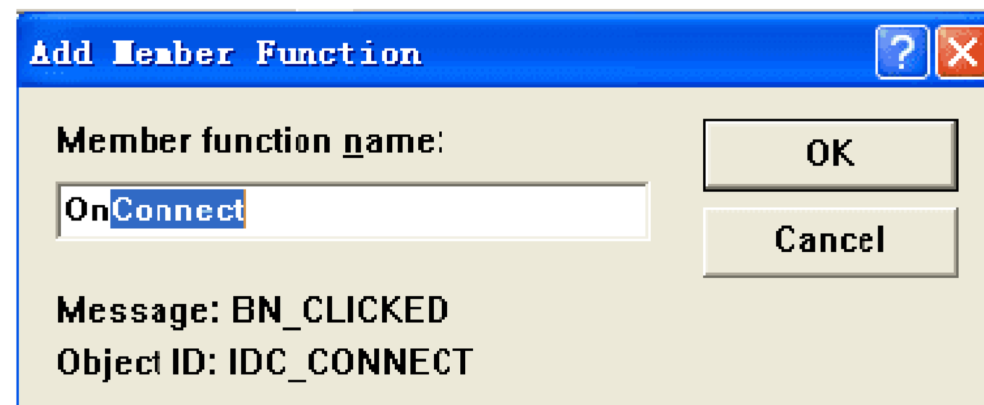


图 2.20　添加成员函数对话框

在该对话框中，用户可以将“连接”按钮的消息响应函数名修改为 OnConnect()。函数代码如下：

```
void CTCPDlg::OnConnect()
{
    CString str,str1;                            //定义字符串
  int port;                                      //定义端口变量
  GetDlgItem(IDC_ADDR)->GetWindowText(str);      //获取服务器地址
  GetDlgItem(IDC_PORT)->GetWindowText(str1);     //获取端口号
  if(str==""||str1=="")                          //判断用户输入是否为 NULL
  {
   MessageBox("服务器地址或端口不能为 NULL");       //显示提示信息
  }
  else
  {
  port=atoi(str1.GetBuffer(1));                  //将端口字符串转换为数字
  addr.sin_family=AF_INET;
  addr.sin_addr.S_un.S_addr=inet_addr(str.GetBuffer(1));
                                                 //转换服务器 IP 地址
  addr.sin_port=ntohs(port);
  GetDlgItem(IDC_TEXT)->SetWindowText("正在连接服务器......\r\n");
                                                 //提示用户正在连接服务器
  if(::connect(s,(sockaddr*)&addr,sizeof(addr))!=SOCKET_ERROR)
                                                 //连接服务器
  {
   GetDlgItem(IDC_TEXT)->GetWindowText(str);     //显示提示信息
   str+="连接服务器成功！\r\n";
   GetDlgItem(IDC_TEXT)->SetWindowText(str);
   GetDlgItem(IDC_SENDTEXT)->EnableWindow(true); //设置控件的显示状态
   GetDlgItem(IDC_SEND)->EnableWindow(true);
   GetDlgItem(IDC_ADDR)->EnableWindow(false);
   GetDlgItem(IDC_PORT)->EnableWindow(false);
  }
  else                                           //连接失败
  {
     GetDlgItem(IDC_TEXT)->GetWindowText(str);
       str+="连接服务器失败！请重试\r\n";
     GetDlgItem(IDC_TEXT)->SetWindowText(str);
  }
```

```
}
}
```

将上面的代码保存以后，进行编译并运行。如果客户端连接服务器成功，则程序会提示用户连接成功，如图 2.21 所示。否则，程序提示用户连接服务器失败，如图 2.22 所示。



图 2.21　客户端连接服务器成功

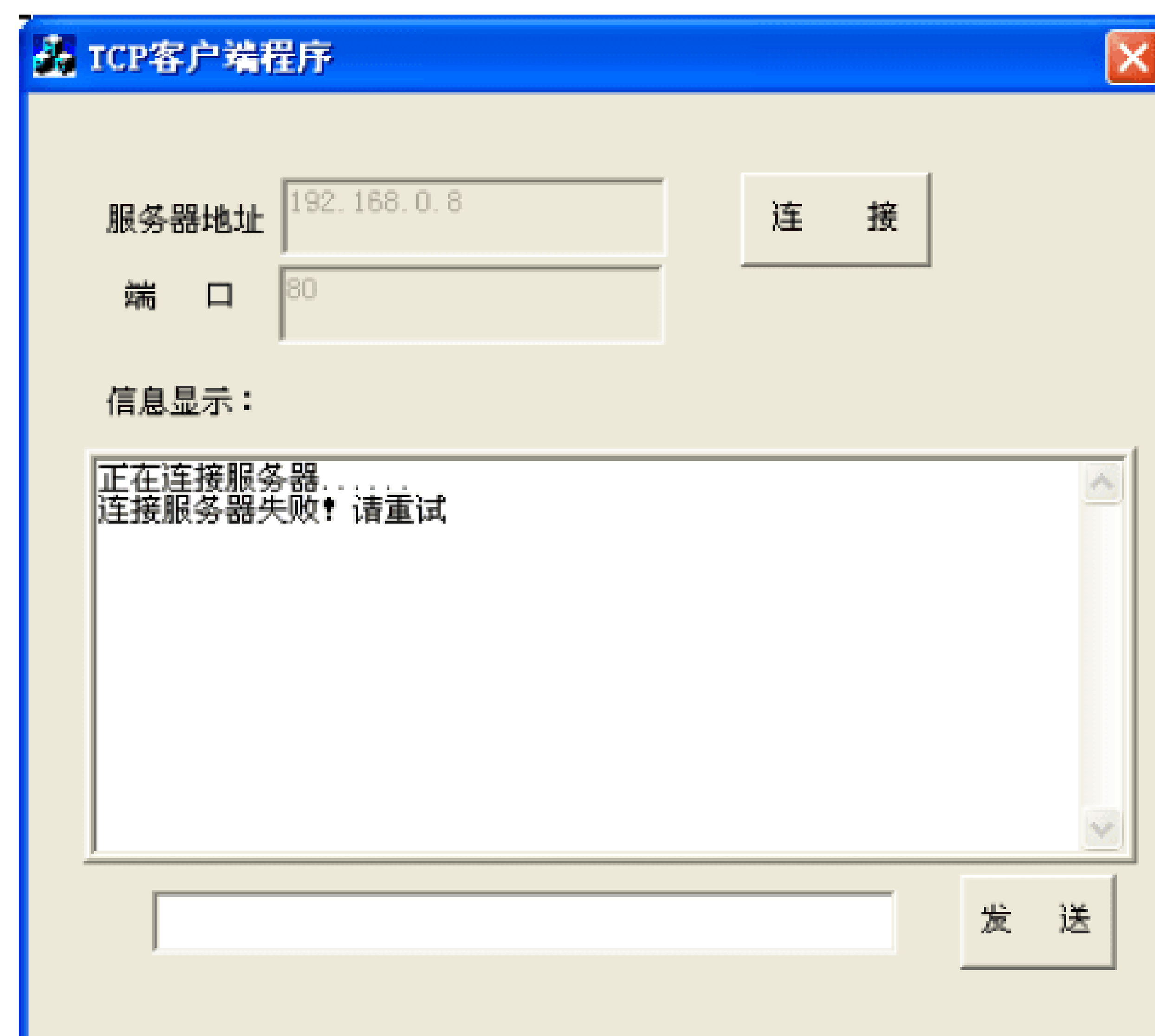


图 2.22　客户端连接服务器失败

当客户端与服务器连接成功之后，用户便可以发送消息到服务器了。现在，用户需要为“发送”按钮添加相应的消息响应函数，并指定该函数名为 OnSend()。该函数相关代码如下：

```
void CTCPDlg::OnSend()
{
CString str,str1;                                   //定义字符串变量
GetDlgItem(IDC_SENDTEXT)->GetWindowText(str);       //获取用户发送的消息字符串
if(str=="")                                         //不允许用户发送空消息
{
    GetDlgItem(IDC_TEXT)->GetWindowText(str1);      //获取信息框中的内容
    str1+="\r\n";                                   //添加回车换行符
    str1+="消息不能为空\r\n";
    GetDlgItem(IDC_TEXT)->SetWindowText(str1);      //设置信息框中的内容
}
else
{
    ::send(s,str.GetBuffer(1),sizeof(str),0);       //发送信息到指定服务器
    GetDlgItem(IDC_TEXT)->GetWindowText(str1);      //获取信息框中的内容
    str1+="\r\n";                                   //添加回车换行符
    str1+=str;
    GetDlgItem(IDC_TEXT)->SetWindowText(str1);      //设置信息框中的内容
}
}
```

在代码中，用户通过调用函数 send()将消息发送到指定的服务器，并将该消息显示在本地的信息显示框中，如图 2.23 所示。

作为客户端，还应该具有接收并显示服务器所发送的消息。在本实例中，将采用异步套接字模式实现该功能。在 VC 中，将套接字设置为异步模式，可以调用函数

WSAAsyncSelect()实现。该函数原型如下：

```
int WSAAsyncSelect (
  SOCKET s,
  HWND hWnd,
  unsigned int wMsg,
  long lEvent
);
```

函数各个参数如下：

- ❑ 参数 s 表示需要设置为异步模式的套接字句柄。
- ❑ 参数 hWnd 表示接收消息响应的窗口句柄。
- ❑ 参数 wMsg 表示响应消息标识。
- ❑ 参数 lEvent 表示发生在该套接字上的事件，取值如表 2.3 所示。

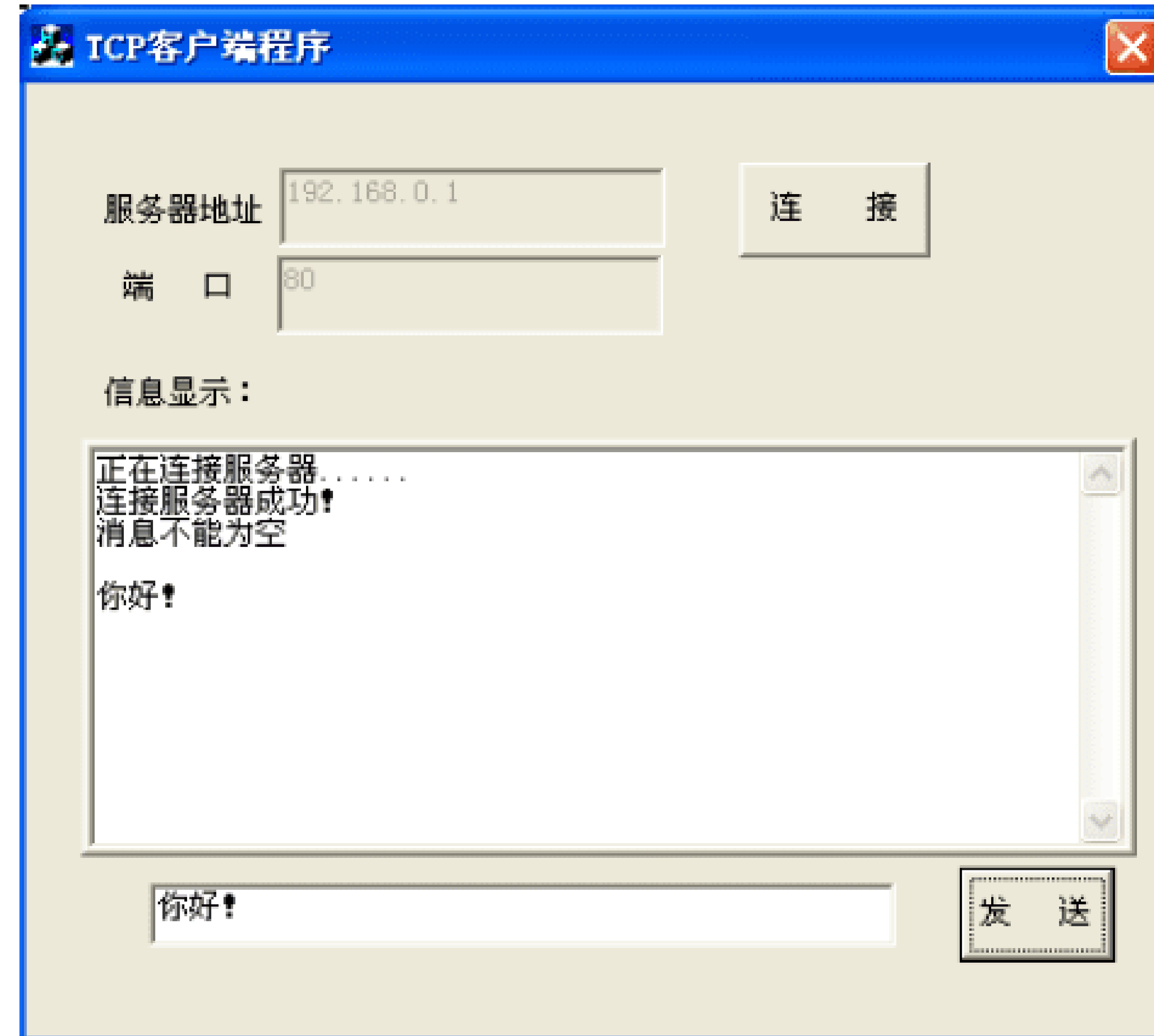


图 2.23　客户端发送消息

表 2.3　套接字事件部分标识及其意义

套接字事件取值	含 义 表 示	套接字事件取值	含 义 表 示
FD_READ	套接字上发生读取事件	FD_ACCEPT	套接字上发生连接事件
FD_WRITE	套接字上发生写入事件	FD_CLOSE	套接字上发生关闭事件

首先，在程序初始化函数 OnInitDialog 中，将套接字设置为异步模式。代码如下：

```
BOOL CTCPDlg::OnInitDialog()
{
    ...                                           //省略部分代码
GetDlgItem(IDC_TEXT)->EnableWindow(false);        //设置各个控件的显示状态
GetDlgItem(IDC_SENDTEXT)->EnableWindow(false);
GetDlgItem(IDC_SEND)->EnableWindow(false);
s=::socket(AF_INET,SOCK_STREAM,0);                //创建套接字
    :: WSAAsyncSelect(s,this->m_hWnd,WM_SOCKET, FD_READ);
                                                  //将套接字设置为异步模式
return TRUE;
}
```

代码中，将异步套接字处理的时间指定为读取事件 FD_READ，并且将该事件的处理消息指定为 WM_SOCKET。该消息是在 CTCPDlg 类头文件中定义的自定义消息。代码如下：

```
#define WM_SOCKET WM_USER+100                     //定义自定义消息
class CTCPDlg : public CDialog
{
  ...                                             //省略部分代码
  protected:
afx_msg void OnSocket(WPARAM wParam,LPARAM lParam);//自定义消息响应函数
    }
```

用户自定义消息以及该消息的响应函数成功后，还需要在消息映射表中将消息与响应函数相关联。代码如下：

```
BEGIN_MESSAGE_MAP(CTCPDlg, CDialog)
    //{{AFX_MSG_MAP(CTCPDlg)
    ON_BN_CLICKED(IDC_CONNECT, OnConnect)
    ON_BN_CLICKED(IDC_SEND, OnSend)
    ON_MESSAGE(WM_SOCKET,OnSocket)                //自定义消息映射项
    //}}AFX_MSG_MAP
END_MESSAGE_MAP()
```

最后，在自定义消息响应函数 OnSocket()中，实现套接字事件的处理。代码如下：

```
void CTCPDlg::OnSocket(WPARAM wParam,LPARAM lParam)
{
 char cs[100]={0};                               //定义数据缓冲区
if(lParam==FD_READ)                              //如果是套接字读取时间
{
 CString num="";                                 //定义字符串变量
 recv(s,cs,100,NULL);                            //接收数据
 GetDlgItem(IDC_TEXT)->GetWindowText(num);       //获取消息框中的内容
 num+="\r\n 服务器说: ";                          //添加回车换行符
 num+=(LPTSTR)cs;                                //将接收到的数据转换为字符串
    GetDlgItem(IDC_TEXT)->SetWindowText(num);    //设置消息框内容
}
}
```

由于在本实例中，仅处理了套接字的读取事件，所以使用了代码“if(lParam==FD_READ)”。如果用户需要处理的套接字事件比较多，那么应该在代码中使用关键字 switch 进行分类判断。程序运行效果如图 2.24 所示。

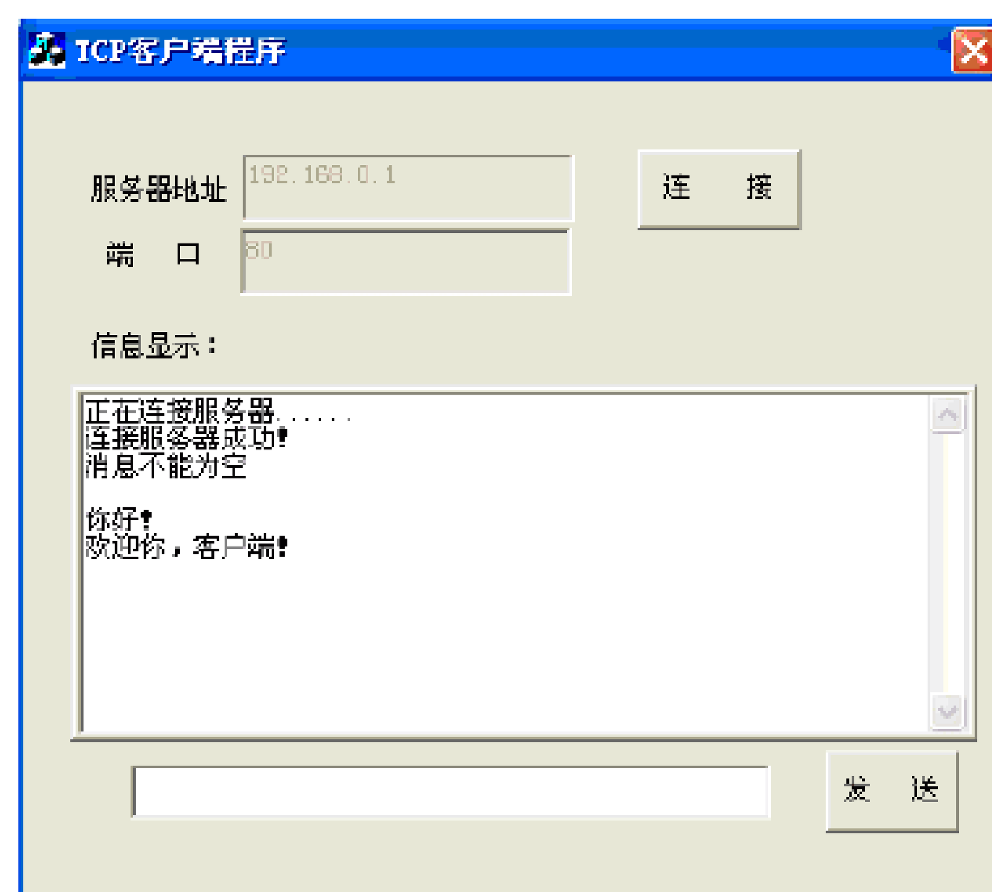


图 2.24　程序运行效果

到这里，用户基本上完成了客户端应有的功能。在客户端程序中，需要用户注意连接服务器之前，必须首先知道服务器的 IP 地址等相关信息。否则，程序将无法正确连接到服务器。

2.3.2　TCP 服务器程序

在 2.3.1 节中，已经向用户讲解了制作 TCP 客户端程序的相关方法。所以，在本节中将向用户继续讲解在 VC 中怎样制作 TCP 服务器程序。

1. 创建工程

在 VC 中，创建 TCP 服务器工程的步骤与创建 TCP 客户端工程的步骤一样，只是在修改工程名称时应该为“TCP 服务器程序”，如图 2.25 所示。

其他相关设置步骤均与 TCP 客户端工程的设置步骤一样。所以，在本章中不再对此内容进行讲述，请用户复习上一节中的内容。

注意：用户在 VC 中创建实例工程的步骤大体相同。

2．界面设计

服务器工程创建完成之后，用户可以打开资源管理器中的对话框资源进行界面的设计。本实例中，为了完成服务器的基本功能，所以在对话框面板上添加如表 2.4 所示的控件，并调整其位置以及大小。



图 2.25　创建 TCP 服务器程序工程

表 2.4　控件ID、属性以及作用

控件 ID	控 件 类 型	控件作用描述
IDC_TEXT	编辑框	显示发送与接收到的信息
IDC_SENDTEXT	编辑框	输入发送的字符串
IDC_SEND	按钮	发送信息
IDC_ADDR	静态	显示本机地址

用户将表 2.4 中所示控件添加到对话框面板中后，应该调整各个控件的位置以及大小，达到界面的美化。运行之后的程序界面效果，如图 2.26 所示。

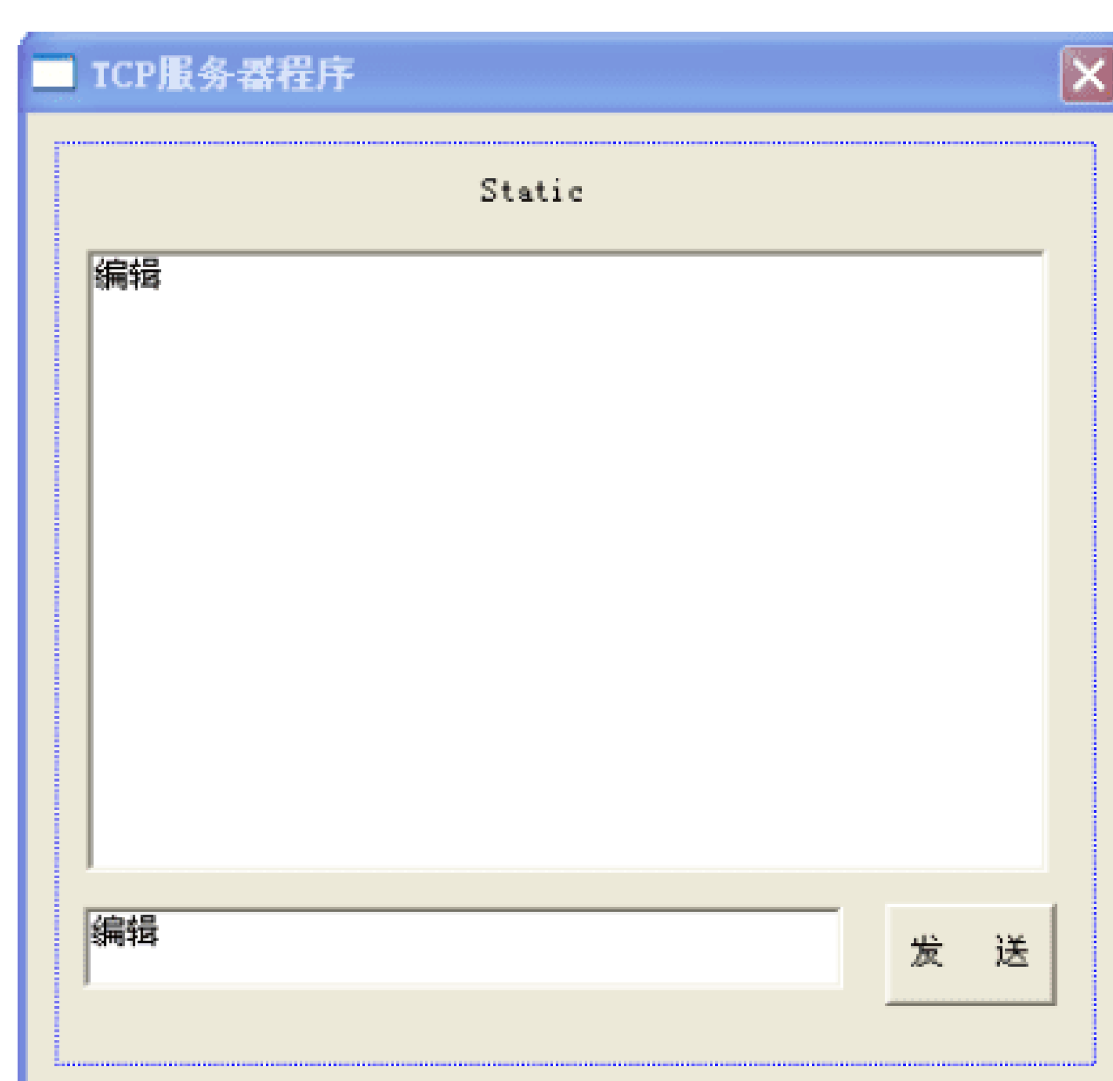


图 2.26　程序界面效果

3．界面初始化

与 TCP 客户端一样，服务器程序启动时也需要界面初始化。不过，服务器界面在初始化时，还应该同时完成套接字的创建以及地址绑定等处理工作。首先，定义套接字相关变量。代码如下：

```
class CTCPDlg : public CDialog
{
public:
 CTCPDlg(CWnd* pParent = NULL);
SOCKET s,s1;                                //定义套接字句柄
sockaddr_in addr,add1;                      //定义套接字地址结构变量
...                                         //省略部分代码
   }
```

然后，在对话框初始化函数中创建套接字并且将套接字绑定到本地地址。代码如下：

```
BOOL CTCPDlg::OnInitDialog()
{
 CDialog::OnInitDialog();
    addr.sin_family=AF_INET;                            //初始化套接字地址结构
 addr.sin_port=htons(80);                               //指定端口
 addr.sin_addr.S_un.S_addr=INADDR_ANY;
    ::bind(s,(sockaddr*)&addr,sizeof(addr));            //绑定本地地址
 ::listen(s,2);                                         //监听端口
 GetDlgItem(IDC_TEXT)->EnableWindow(false);             //禁用消息显示框
 GetDlgItem(IDC_ADDR)->SetWindowText("服务器监听已经启动！");
 return TRUE;
}
```

将以上代码保存并编译运行以后，服务器界面初始化运行后的效果，如图 2.27 所示。

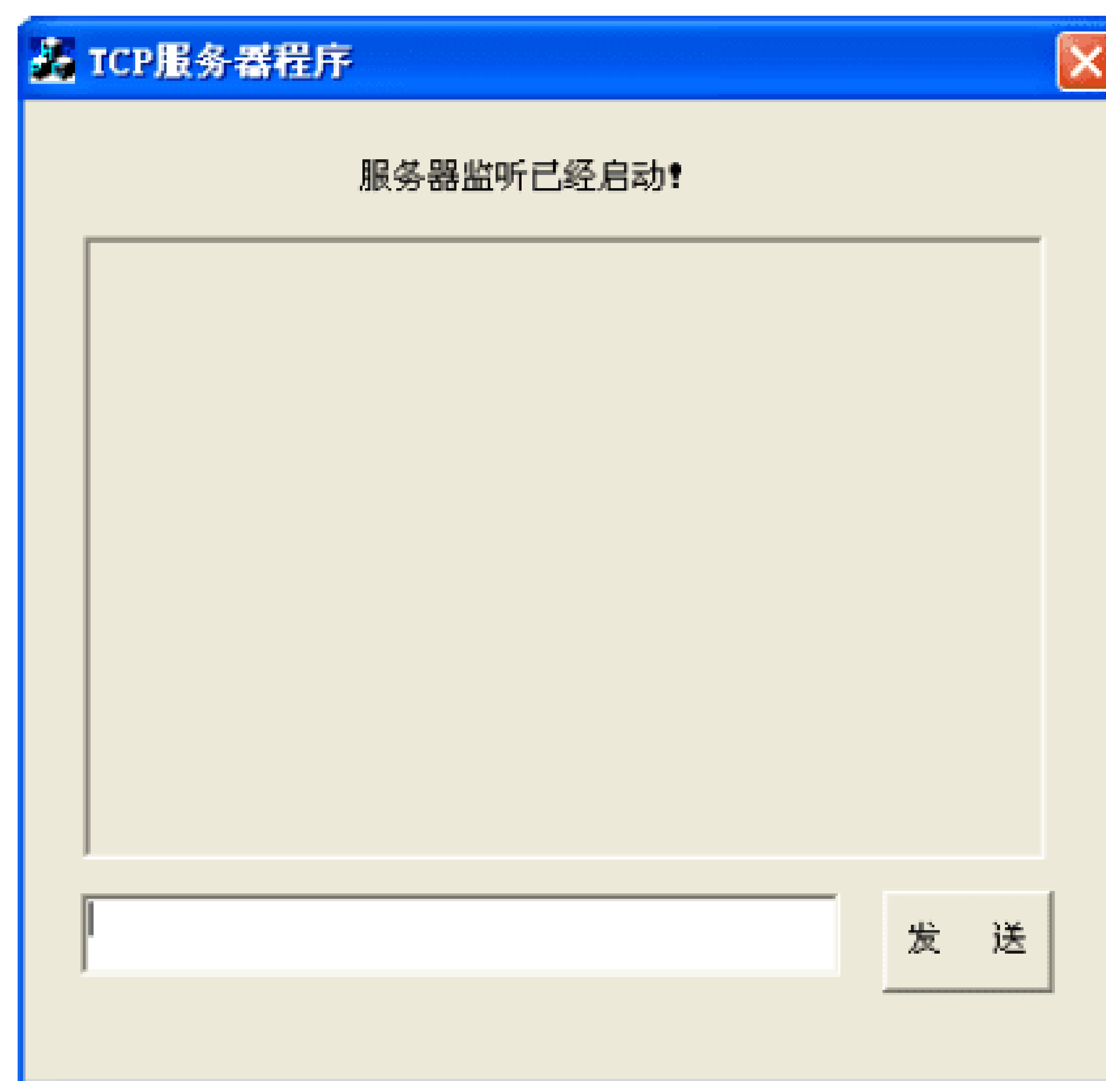


图 2.27　服务器界面初始化

4．功能实现

TCP 服务器应该具有监听、发送和接收数据的功能。首先，用户应该为程序添加监听功能。因为，服务器必须等待客户端的连接请求到来之后，才能实现接收和发送数据。将服务器创建的套接字设置为异步模式，并将套接字事件设置为连接和读取事件。代码如下：

```
#define WM_SOCKET WM_USER+1000                          //自定义套接字消息
class CTCPDlg : public CDialog
{
   ...                                                  //省略部分代码
protected:
 afx_msg HCURSOR OnQueryDragIcon();
 afx_msg void OnSocket(WPARAM wParam,LPARAM lParam);    //自定义消息响应函数
 }
BOOL CTCPDlg::OnInitDialog()                            //对话框初始化函数
{
 CDialog::OnInitDialog();
    addr.sin_family=AF_INET;                            //填充套接字地址结构
 addr.sin_port=htons(80);
 addr.sin_addr.S_un.S_addr=INADDR_ANY;
    ::bind(s,(sockaddr*)&addr,sizeof(addr));            //绑定本地地址
 ::listen(s,2);                                         //监听端口
::WSAAsyncSelect(s,this->m_hWnd,WM_SOCKET,FD_ACCEPT|FD_READ);
                                                        //设置异步套接字
 return TRUE;
}
```

当服务器监听套接字上有相关的事件发生时，程序便会调用自定义的消息响应函数 OnSocket()对该事件进行处理。代码如下：

```
void CTCPDlg::OnSocket(WPARAM wParam,LPARAM lParam)
{
 CString str13;                                         //定义字符串变量
 char cs[100]={0};                                      //定义字符数组
switch (lParam)
{
     case FD_ACCEPT:                                    //如果事件为连接事件
```

```
        {
    s1=::accept(s,(SOCKADDR*)&addl,(int*)sizeof(addl));
                                                        //应答客户端的连接请求
      n=n+1;                                            //连接的客户端数目
      str13.Format("有%d 客户已经连接上了",n);            //格式化字符串
        GetDlgItem(IDC_TEXT)->SetWindowText(str13);     //设置消息框内容
     str13+=::inet_ntoa(addl.sin_addr);                 //转换 IP 地址
     str13+="登录\r\n";
     GetDlgItem(IDC_TEXT)->SetWindowText(str13);        //设置显示消息
                                   }
        break;                                          //跳出该函数

    case FD_READ:                                       //处理读取事件
          {
           CString num="";                              //定义字符串
           ::recv(s1,cs,100,0);                         //接收消息
             GetDlgItem(IDC TEXT)->GetWindowText(num);
               num+="\r\n";
             num+=(LPTSTR)::inet ntoa(addl.sin addr);   //转换 IP 地址
             num+="对您说: ";
             num+=(LPTSTR)cs;
             GetDlgItem(IDC TEXT)->SetWindowText(num);
}
        break;
}
}
```

用户在以上代码中，实现了服务器的应答客户端的连接请求以及显示信息等功能。当有客户端向服务器发送连接请求时，服务器将显示相关信息，如图 2.28 所示。如果已经连接的客户端发送消息到服务器，则服务器程序调用函数 recv()接收该信息并将该消息显示在信息框中，如图 2.29 所示。



图 2.28　服务器应答客户端的连接请求



图 2.29　服务器显示接收到的信息

现在，服务器端已经实现了应答服务器连接请求和接收客户端信息的功能。但是，作为服务器还需要具有发送消息的功能。首先，用户可以使用 VC 应用程序向导为“发送”按钮添加消息响应函数，如图 2.30 所示。

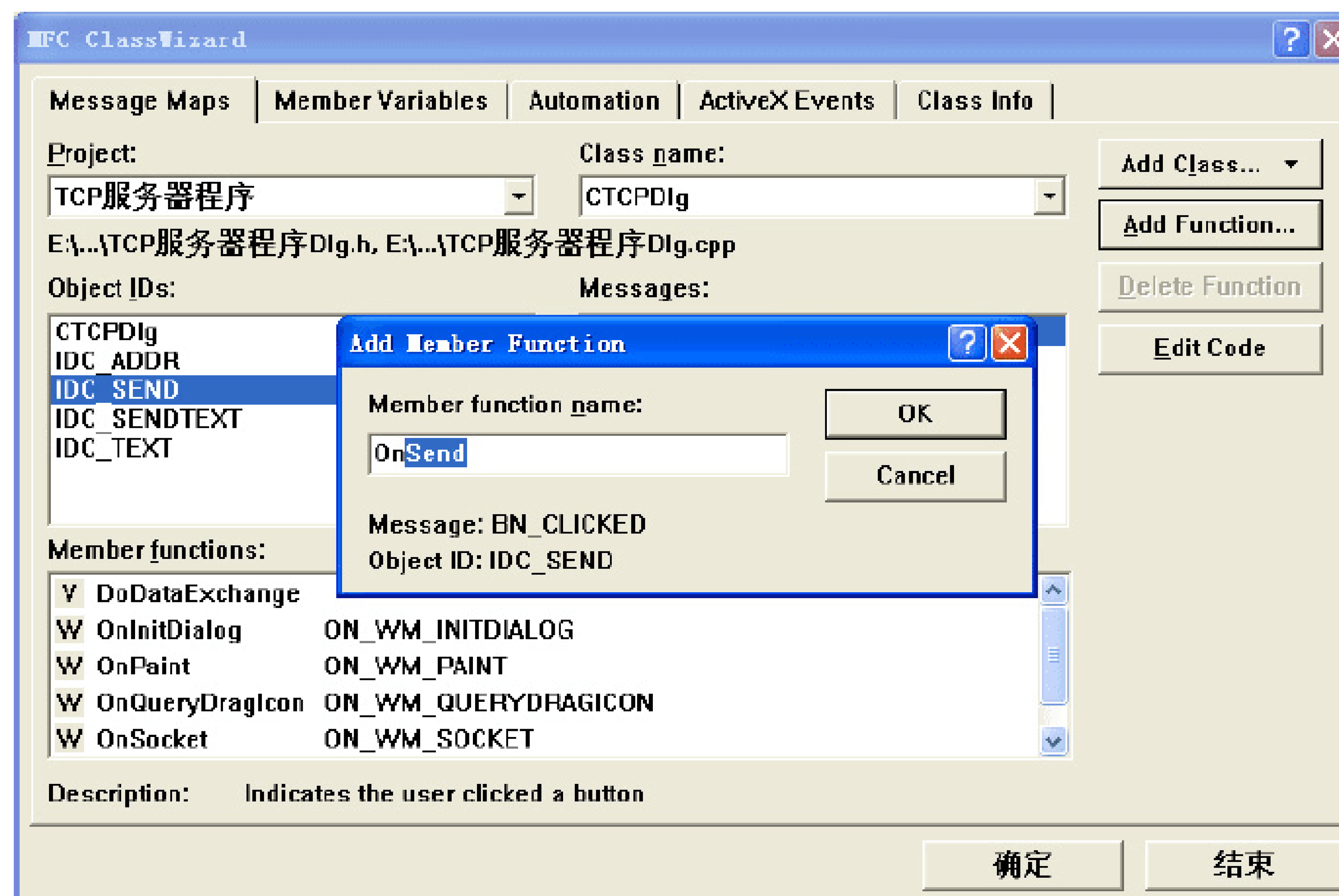


图 2.30　为“发送”按钮添加消息响应函数

用户可以通过 Add Member Funtion 对话框修改消息响应函数的函数名。在该实例中，将该函数名修改为 OnSend()。然后，单击 OK 按钮。

在函数 OnSend()中，服务器程序应该将发送到客户端的消息也显示在服务器端界面中。这样，用户在使用该功能时，对于信息的发送与接收功能的实现的理解比较直观。代码如下：

```
void CTCPDlg::OnSend()
{
 CString str="";                                    //定义字符串
 GetDlgItem(IDC_SENDTEXT)->GetWindowText(str);      //获取发送消息
 if(str==" ")                                       //消息不能为空
 {
    MessageBox("消息不能为空！");                    //显示提示信息
}
else
{
    if(::send(s1,str.GetBuffer(1),sizeof(str),0)!=SOCKET_ERROR)
                                                    //发送消息
    {
    GetDlgItem(IDC_TEXT)->SetWindowText("消息已经发送到客户端！\r\n");
                                                    //在消息框中显示提示信息
      GetDlgItem(IDC_TEXT)->GetWindowText(str);     //获取消息框中的内容
    str+="\r\n";                                    //添加回车换行符
        GetDlgItem(IDC_TEXT)->SetWindowText(str);   //设置消息框中的内容
    }
    else                                            //如果发送消息不成功
    {
    GetDlgItem(IDC_TEXT)->SetWindowText("消息发送失败！\r\n");
                                                    //提示用户发送消息失败
    }
}
}
```

在程序中，用户首先调用函数获取发送消息框中的内容并将其存放在字符串变量 str

中。如果发送的消息是空字符串，则提示用户不能发送空消息。接着，程序调用函数 send()将消息发送到客户端，并将该消息显示在服务器界面中，如图 2.31 所示。

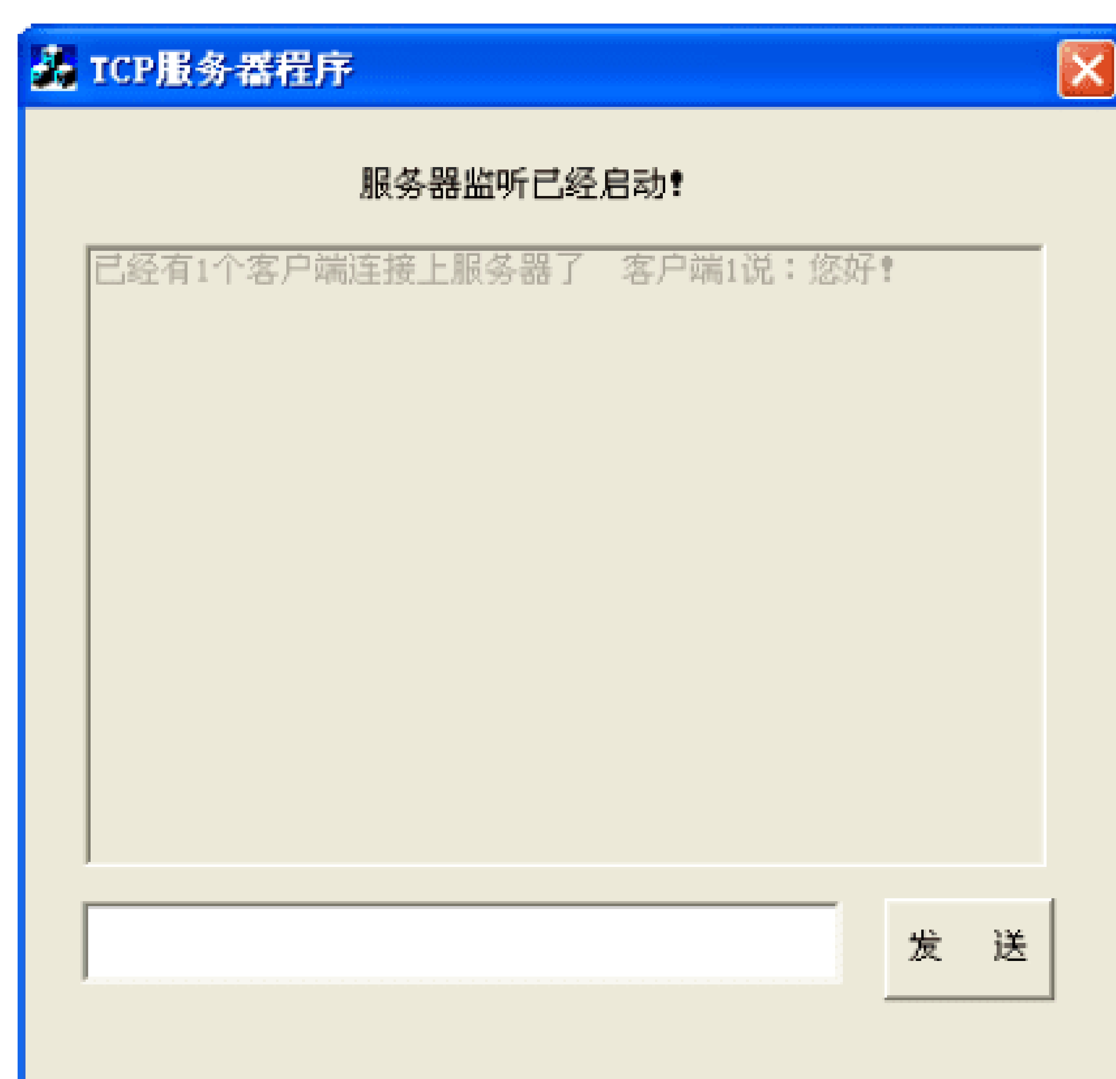


图 2.31　服务器发送并显示信息

在本节中，向用户讲解了在 VC 中开发 TCP 服务器程序的步骤和方法。通过编写实例程序向用户分别讲解了服务器工程的创建，构建服务器界面以及服务器各个功能的实现等。如果用户希望进一步学习 TCP 服务器编程，用户就可以在实例程序的基础上进行修改，以便达到更好的学习效果。

2.4　小　　结

在本章中，主要向用户介绍了 Socket 套接字编程中需要使用的基础知识以及相关函数。在介绍套接字相关函数时，主要讲解了函数的原型以及使用等。在 2.2 节和 2.3 节中，通过在 VC 中创建实例工程向用户分别介绍了创建工程，设置工程和实例程序编写的方法。用户通过本章的学习，应该掌握基本的套接字编程方法以及使用 VC 编译器创建工程等操作。

在本章实例中，所有代码均在随书光盘的对应章节中，用户可以通过本书中所讲述的理论知识结合光盘中的实例代码进行学习。这样，可提高用户的学习效率。

第 3 章　多线程与异步套接字编程

在 Windows 操作系统中，线程是指系统中最小的功能执行单元，其可以独立地完成某一项功能。所以在进行 Windows 编程中，如果用户使用多线程处理某个功能，那么该功能被处理的效率远比单个线程处理的效率高。在本章中，将向用户介绍使用多线程处理异步套接字编程的相关方法。

3.1　多线程技术

在 Windows 操作系统中，所有程序的功能都是由每个程序中的多个线程共同完成。从某种特定的意义上而言，线程才是计算机真正意义上的功能执行者。而从线程执行的数目而言，线程可以分为单线程和多线程。其中，多线程是由多个单线程组成。如果从线程的执行效率而言，多线程比单线程的执行效率高很多。那么，当用户在编程时，使用多线程技术可以提高程序的执行效率。

3.1.1　基本概念

在本节中，将介绍一些关于计算机进程和线程方面的基本概念。用户通过这些基本概念的学习，将学习到计算机程序的工作原理以及多线程处理方面的基础知识。

1．计算机进程

在计算机操作系统中，进程是指当可执行文件运行时，系统所创建的内核对象。例如，在计算机中，用户可以通过任务管理器查看当前系统中所有的进程，如图 3.1 所示。



图 3.1　显示系统中所有的进程

在一个以“.EXE”为后缀名的可执行程序中，可以包括一个或多个进程，并且每个进程都有自己的执行地址空间。这些地址空间在逻辑层面上可以被不同的进程重复使用。例如，计算机系统中有两个进程，分别为进程 A 和进程 B。如果进程 A 在某一地址空间中存放了一个数据，而进程 B 可以在同一地址空间中存放另一个数据。当两个进程同时在该地址空间中取出各自对

应的数据时，程序不会出现非法访问内存等错误信息。这是因为在进程中真正执行某个功能的应该是该进程中的线程，这些线程只是共享同一个进程的地址空间。

2．计算机线程

线程是计算机中最小的执行单元。通常，当 Windows 应用程序运行时，操作系统都会为其自动创建一个线程，即主线程。通过主线程，用户可以创建多个线程或进程。由于一个进程中的所有线程共享该进程地址空间，所以，在同一个进程中可以实现多个线程间的相互通信。

当用户编程时，为了完成某一项功能可以使用多线程技术创建多个线程共同完成这个功能。这种方法比单线程技术实现同一功能的效率快。实际上，现在很多的 CPU 处理器都只支持单线程技术，但是一个多线程程序为什么仍能运行，这是因为系统程序为系统中的每个线程都分配了执行时间，而且这个执行时间非常短，以至于用户感觉几个线程在同时运行。

在本章中，主要是让用户学习多线程技术的编程方法以及它在网络编程中的使用等。所以，关于多线程技术的其他知识，本书将不再进行深入讲解。

3.1.2　创建线程

用户编程时，使用多线程技术需要首先创建线程，然后再使用这些线程执行相应的功能。如果用户是在 VC 中编写多线程程序，则可以调用 API 函数 CreateThread()创建线程。该函数原型如下：

```
HANDLE CreateThread(
  LPSECURITY_ATTRIBUTES lpThreadAttributes,
  DWORD dwStackSize,
  LPTHREAD_START_ROUTINE lpStartAddress,
  LPVOID lpParameter,
  DWORD dwCreationFlags,
  LPDWORD lpThreadId
);
```

该函数的作用是用于创建一个线程，并将返回该线程的句柄。其中，各个参数含义如下：

- 参数 lpThreadAttributes 是一个指向结构体 SECURITY_ATTRIBUTES 的指针，表示指定新建线程的安全属性。该参数可以设置为 NULL，表示创建线程时使用默认的安全属性。
- 参数 dwStackSize 指定线程初始化时地址空间的大小。如果这个参数指定为 0，那么新创建线程的地址空间大小与调用该函数的线程地址空间大小一样。
- 参数 lpStartAddress 将指定该线程的线程函数的地址。当线程创建成功以后，新建线程将调用该线程函数执行某个功能。
- 参数 lpParameter 表示将要传递给新建线程的命令行参数。新建线程可以根据该命令参数的不同而执行不同的功能。
- 参数 dwCreationFlags 用于指定新线程创建后是否立即运行。其取值如表 3.1 所示。

表 3.1　线程创建标记

状　态　值	作　　用
CREATE_SUSPENDED	线程创建成功后暂停运行
0	线程创建成功后立即运行

注意：当用户创建线程时，将该参数值指定为 CREATE_SUSPENDED，则线程将处于暂停状态，直到用户调用相关函数将线程恢复运行为止。

- 参数 lpThreadId 表示新建线程的 ID 号。在这里，用户可以将该参数设置为 NULL。

例如，用户在程序中，使用该函数分别创建两个线程，并在这两个线程中分别打印出各自的函数信息。首先，用户打开 VC，并创建一个基于控制台程序窗口的工程，并将工程名修改为“创建线程”，如图 3.2 所示。



图 3.2　创建控制台工程

然后，单击“确定”按钮，转到工程设置中将该工程设置为一个空的控制台程序，如图 3.3 所示。

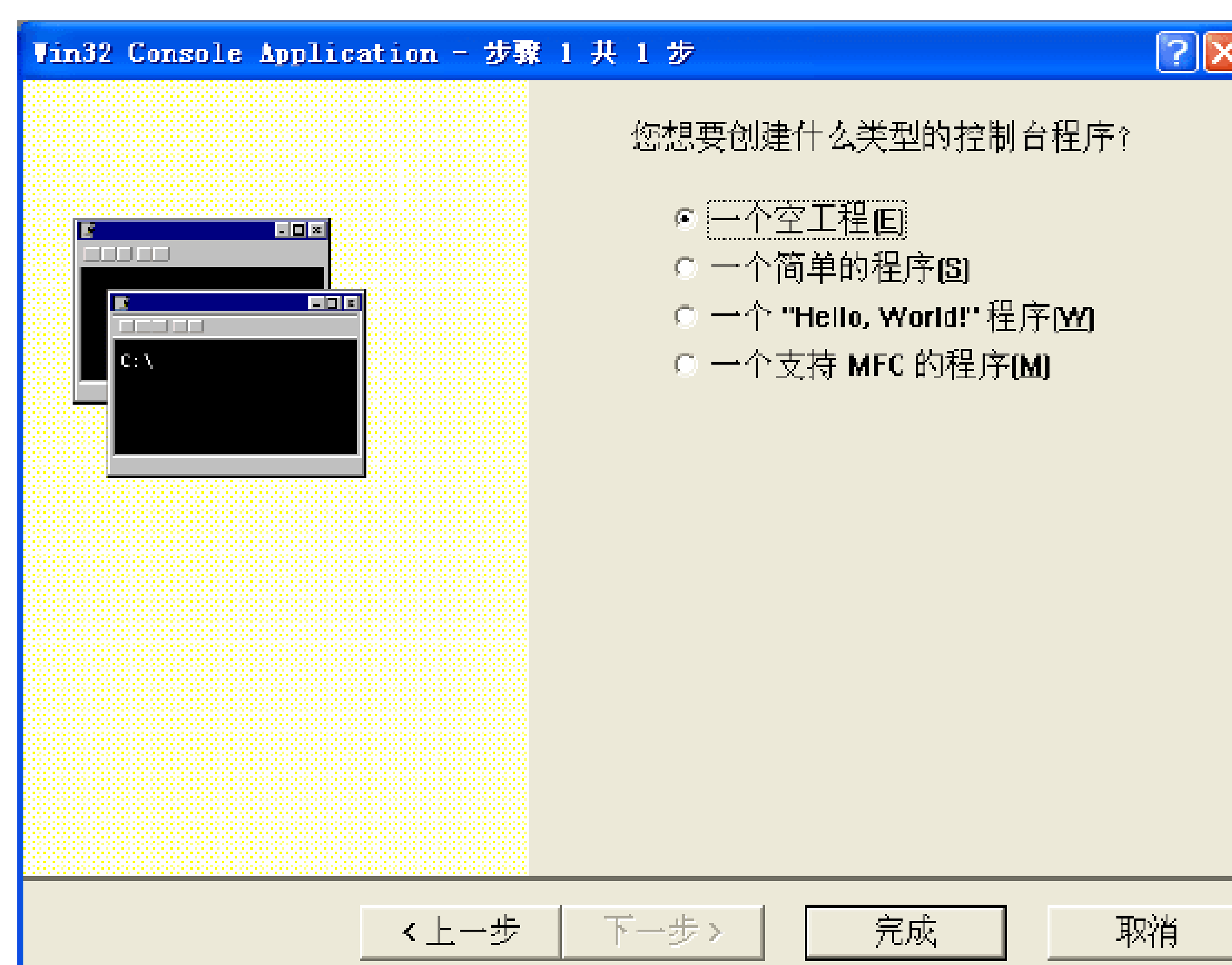


图 3.3　指定该工程为空工程

最后，单击“完成”按钮，完成该控制台工程的设置，返回到 VC 主界面中。用户在该工程中需要添加一个空白的 C++源文件以便编写代码。创建线程实例，代码如下：

```
#include <windows.h>                                       //包含相应头文件
#include <stdio.h>
DWORD WINAPI myfun1(                                       //声明线程函数
  LPVOID lpParameter
);
DWORD WINAPI myfun2(
  LPVOID lpParameter
);
int main()                                                 //主函数
{
 HANDLE h1,h2;                                             //定义句柄变量
 h1=::CreateThread(NULL,0,myfun1,NULL,0,NULL);             //创建线程 1
 printf("线程 1 开始运行！\r\n");                          //输出线程 1 运行信息
 h2=::CreateThread(NULL,0,myfun2,NULL,0,NULL);             //创建线程 2
 printf("线程 2 开始运行！\r\n");                          //输出线程 2 运行信息
 ::CloseHandle(h1);                                        //关闭线程句柄对象
 ::CloseHandle(h2);
     if(getchar()=='q')                                    //如果用户输入字符 q
{
    return 0;                                              //程序正常退出
 }
 else                                                      //如果用户输入的字符不是 q
 {
    ::Sleep(100);                                          //程序睡眠
 }
}
DWORD WINAPI myfun1(LPVOID lpParameter)                    //分别实现线程函数
{
 printf("线程 1 正在运行！\r\n");                          //输出信息
 return 0;                                                 //正常结束线程函数
}
DWORD WINAPI myfun2(LPVOID lpParameter)
{
 printf("线程 2 正在运行！\r\n");
 return 0;
}
```

在程序中，用户首先声明了两个线程函数，然后在主函数中创建两个线程，分别为 h1 和 h2。当线程创建成功以后，系统会自动调用相应的线程函数并执行其中的功能。将上面的程序在 VC 中进行编译、运行，如图 3.4 所示。



图 3.4　创建线程实例

用户可以从该实例程序中，学习到怎样声明和定义线程函数以及使用函数 CreateThread()进行线程的创建。

注意：用户从程序运行的结果中可以得知，线程 1 和线程 2 并没有按照代码的运行顺序而执行其功能。这时候，用户需要使用线程的同步技术避免类似情况的发生。在 3.2 节中，将为用户讲解实现线程同步的编程方法。

3.2 实现线程同步

线程同步是指同一进程中的多个线程互相协调工作达到一致性。当用户编写程序时，有时会使多个代码段同时读取或修改相同地址空间中的共享数据。此时，在操作系统中，可能会出现一个代码段在读取数据，而另一个代码段却正在修改数据。这样的情况会导致程序发生读写错误，造成程序异常退出。用户为了避免出现类似情况，需要使用线程同步技术。即当一个线程程序对资源进行读写时，其他的线程程序则处于等待状态。

3.2.1 临界区对象

临界区对象是指当用户使用某个线程访问共享资源时，必须使代码段独享该资源，不允许其他线程程序访问该资源。待该代码段访问完资源后，其他程序才能对资源进行访问。这样的模式好比某个用户在试衣间里试衣服，而其他用户则只能等待。当试衣间里的用户出来之后，其他用户才能进入试衣间内。在本节中，将向用户分别介绍如何使用 API 函数和 MFC 类对临界区对象进行编程的方法。

1. 使用API函数操作临界区

当用户在实际编写程序时，使用临界区对象前必须对临界区进行初始化。在 VC 中进行编程，用户可以调用函数 InitializeCriticalSection()对临界区对象进行初始化。该函数原型如下：

```
Void InitializeCriticalSection(
  LPCRITICAL SECTION lpCriticalSection
);
```

该函数的作用是为应用程序初始化临界区。其中，参数 lpCriticalSection 是指向结构体 CRITICAL_SECTION 的指针变量。由于该参数所标识的结构体对象是由操作系统自动进行维护的，所以用户在编程时可以不用理会该结构体变量的具体操作。例如，用户调用该函数对临界区进行初始化。代码如下：

```
CRITICAL_SECTION m_sec;                //定义结构体 CRITICAL_SECTION 变量
InitializeCriticalSection(&m_sec);     //初始化临界区
...                                    //省略部分代码
```

在程序中，用户首先定义了结构体 CRITICAL_SECTION 的变量 m_sec，其用于存放临界区的相关信息。然后，调用函数 InitializeCriticalSection()对临界区进行初始化。

当用户对临界区进行初始化以后，程序便可以进入该临界区并拥有该临界区对象的所有权。而其他程序则只能等待进入临界区的程序释放临界区的所有权后，才能进入临界区进行操作。用户实现这个功能可以调用函数 EnterCriticalSection()。该函数原型如下：

```
Void EnterCriticalSection(
  LPCRITICAL SECTION lpCriticalSection
);
```

该函数的作用是使调用该函数的线程程序进入已经初始化的临界区，并拥有该临界区的所有权。如果线程程序获得临界区的所有权成功，则该函数将返回，调用线程继续执行。否则，该函数将一直等待，这样会造成该函数的调用线程也一直等待。

注意： 当用户使用临界区对象实现线程同步编程时，不应使函数 EnterCriticalSection()等待的时间过长。因为这样会造成调用线程的等待，使操作系统出现假死现象。

例如，用户调用该函数进入临界区操作被保护的共享资源，同时获取该共享资源的所有权。代码如下：

```
...                                     //省略部分代码
EnterCriticalSection(&m_sec);           //进入已经被初始化的临界区
...                                     //省略部分代码
```

如果调用该函数的线程成功获得临界区所有权，那么该线程将继续执行代码。否则，该线程将一直等待，直到获得临界区的所有权。

当线程使用完共享资源后，则必须离开临界区并释放对该临界区的所有权，以便让其他线程也获得访问该共享资源的机会。函数 LeaveCriticalSection()的作用是使已经进入临界区的线程释放对该临界区的所有权并离开临界区。该函数原型如下：

```
void LeaveCriticalSection(
  LPCRITICAL_SECTION lpCriticalSection
);
```

调用该函数的线程将离开指定的临界区并释放该临界区的所有权。当用户使用临界区对象进行编程，一定要在程序不使用临界区时，调用该函数释放临界区所有权。否则，程序将一直等待，造成程序假死。例如，当用户使用完被临界区保护的共享资源后，需要调用函数 LeaveCriticalSection()释放该临界区的所有权。代码如下：

```
...                                     //省略部分代码
LeaveCriticalSection(&m_sec);           //释放临界区所有权并离开该临界区
...                                     //省略部分代码
```

如果调用线程释放临界区的所有权之后，用户应该在程序中调用函数 DeleteCritical Section()将该临界区从内存中删除。该函数原型如下：

```
Void DeleteCriticalSection(
  LPCRITICAL_SECTION lpCriticalSection
);
```

该函数的作用是删除程序中已经被初始化的临界区。如果函数调用成功，则程序会将内存中的临界区删除，防止出现内存错误。例如，用户使用临界区相关的 API 函数进行编程。代码如下：

```
#include <windows.h>                            //包含头文件
#include <stdio.h>
DWORD WINAPI myfun1(                            //声明线程函数
  LPVOID lpParameter
);
DWORD WINAPI myfun2(
  LPVOID lpParameter
);
```

```
static int a1=0;                                    //定义全局变量并初始化
CRITICAL_SECTION Section;                           //定义临界区对象
int main()                                          //主函数
{
 HANDLE h1,h2;                                      //定义线程句柄
 h1=::CreateThread(NULL,0,myfun1,NULL,0,NULL);  //创建线程 1
 printf("线程 1 开始运行！\r\n");
 h2=::CreateThread(NULL,0,myfun2,NULL,0,NULL);  //创建线程 2
 printf("线程 2 开始运行！\r\n");
 ::CloseHandle(h1);                                 //关闭线程句柄对象
 ::CloseHandle(h2);
 InitializeCriticalSection(&Section);               //初始化临界区对象
::Sleep(10000);                                     //程序睡眠 10 秒
printf("正常退出程序请按'q'\r\n");
 if(getchar()=='q')                                 //如果用户输入字符 q
 {
    DeleteCriticalSection(&Section);                //删除临界区对象
 }
 else                                               //如果用户输入的不是 q
 {
    return 0;                                       //程序自动退出
 }
}
DWORD WINAPI myfun1(LPVOID lpParameter)             //线程函数 1
{

 while(1)
 {
     EnterCriticalSection(&Section);                //进入临界区
     a1++;                                          //变量自加
     if(a1<10000)                                   //设置变量 a1 小于 10000
     {
     ::Sleep(1000);                                 //程序睡眠 1 秒
 printf("线程 1 正在计数%d\r\n",a1);
 LeaveCriticalSection(&Section);                    //离开临界区
         }
     else                                           //如果变量大于 10000
     {
     LeaveCriticalSection(&Section);                //离开临界区
         break;                                     //跳出循环
     }
}
return 0;
}
DWORD WINAPI myfun2(LPVOID lpParameter)             //线程函数 2
{

 while(1)
 {
         EnterCriticalSection(&Section);            //进入临界区
     a1++;
     if(a1<10000)
     {
         ::Sleep(1000);                             //程序睡眠 1 秒
 printf("线程 2 正在计数%d\r\n",a1);
 LeaveCriticalSection(&Section);                    //离开临界区
```

```
        }
        else
        {
    LeaveCriticalSection(&Section);
            break;
        }
    }
    return 0;                                   //线程函数返回 0
}
```

在程序中，用户首先声明临界区对象 Section 和两个线程函数（myfun1()和 myfun2()），然后在主函数中创建两个线程，即 h1 和 h2。主线程调用函数 InitializeCriticalSection()初始化临界区对象并通过函数 Sleep()暂停 10 秒。在线程函数 1 中，程序调用函数 EnterCriticalSection()进入临界区，通过循环使变量 a1 自加，然后输出结果并离开临界区将共享变量 a1 的所有权交给线程函数 2。在线程函数 2 中实现同样的功能。程序运行结果，如图 3.5 所示。

用户从程序运行的结果中可以看到，线程 1 和线程 2 的线程函数交替执行输出结果并且变量 a1 的值是按照顺序增加的。由于当每个线程进入临界区中操作共享变量时，另一个线程则只能等待。所以，该程序实现了同进程的不同线程之间的相互协调工作。

如果用户将临界区相关代码注释起来，再编译执行程序，如图 3.6 所示。

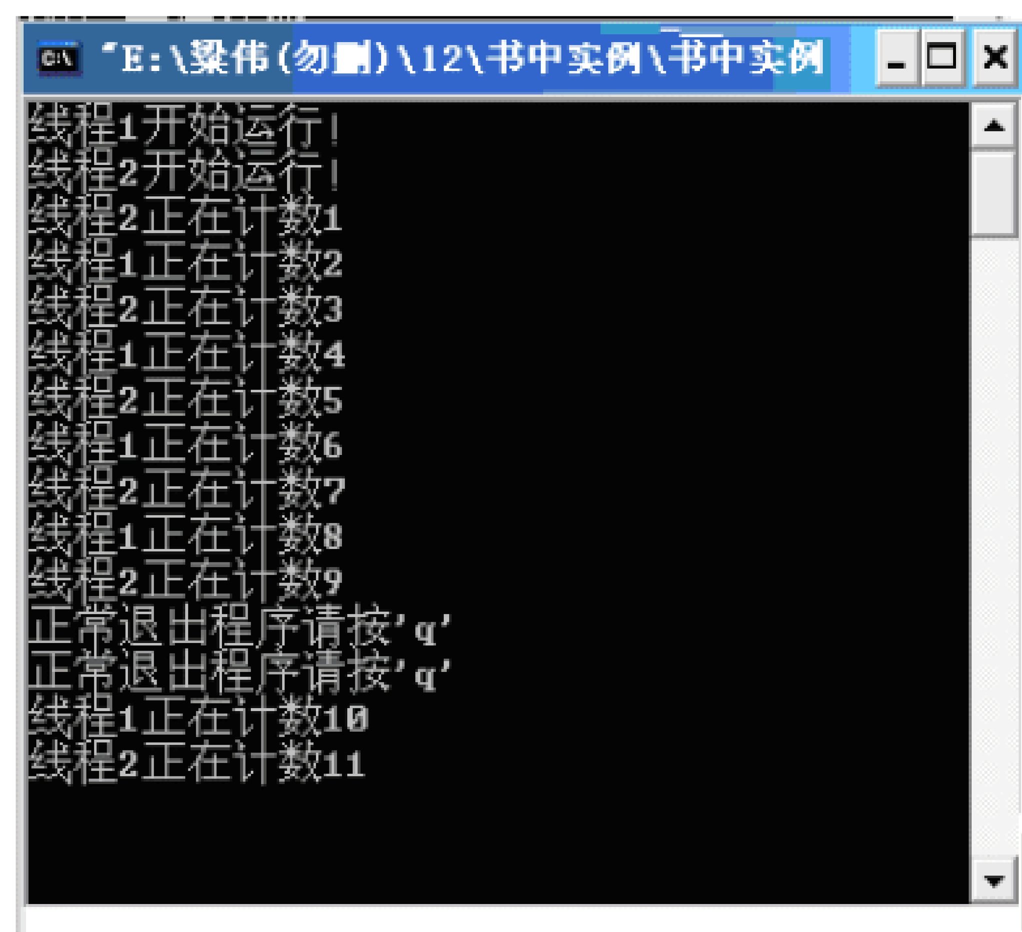


图 3.5　程序运行结果

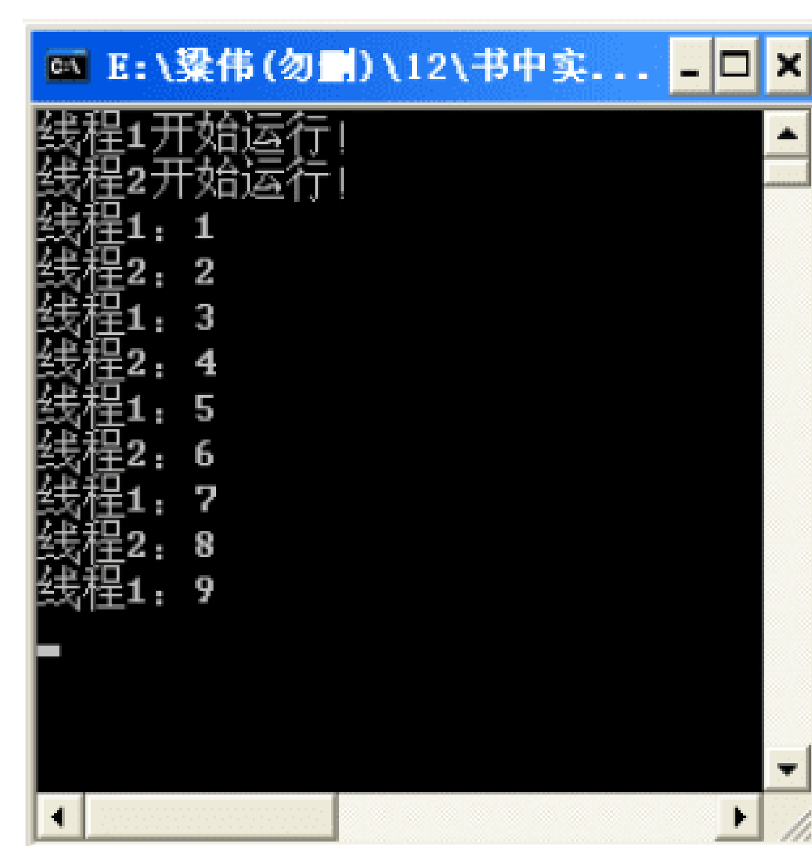


图 3.6　不含临界区对象的程序运行界面

在图 3.6 中，用户会发现两个线程函数并没有交替执行，而且输出的变量结果也未按照顺序增加。所以，在线程同步中临界区对象是非常重要的。

2. 使用CCriticalSection类操作临界区

CCriticalSection 类是 MFC 中所定义的临界区类，其作用与临界区相关 API 函数实现的功能一样。本节中，将向用户简要介绍该类在实际编程中的成员函数以及用法。

首先，用户编程时为了方便线程函数访问 CCriticalSection 类对象，必须将该对象定义为全局变量。代码如下：

```
CCriticalSection m_Sec;          //定义全局变量 m_Sec
main()
{
...                              //省略部分代码
```

```
  }
```

然后，调用该类中成员函数 Lock()对临界区进行锁定。其原型如下：

```
BOOL Lock();                          //锁定临界区
```

函数 Lock()的作用是程序进入临界区执行相关功能并获得该临界区的所有权。如果函数调用成功，则返回 true。否则，函数返回 false。例如，程序根据该函数的返回值判断锁定临界区是否成功。代码如下：

```
...                                    //省略部分代码
if(m_Sec.Locket())                     //调用函数锁定临界区
{
  MessageBox("程序锁定临界区成功！");  //提示信息
    }
else                                   //如果锁定失败
{
 MessageBox("程序锁定临界区失败！");
     }
...                                    //省略部分代码
```

如果程序不再使用临界区，可以调用成员函数 Unlock()离开临界区并释放其所有权。该函数原型如下：

```
virtual BOOL Unlock();                 //释放临界区
```

该函数调用成功，则返回 true。否则，函数将返回 false。例如，用户使用 Ccritical Section 类进行临界区编程。代码如下：

```
#include <windows.h>                                        //包含头文件
#include <stdio.h>
DWORD WINAPI myfun1(                                        //声明线程函数
  LPVOID lpParameter
);
DWORD WINAPI myfun2(
  LPVOID lpParameter
);
CCriticalSection m Sec;                                     //定义全局变量 m Sec
int a=0;                                                    //定义全局变量 a
main()
{
 HANDLE h1,h2;                                              //定义线程句柄
 h1=::CreateThread(NULL,0,myfun1,NULL,0,NULL);              //创建线程 1
 printf("线程 1 开始运行！\r\n");
 h2=::CreateThread(NULL,0,myfun2,NULL,0,NULL);              //创建线程 2
 printf("线程 2 开始运行！\r\n");
 ::CloseHandle(h1);                                         //关闭线程句柄对象
 ::CloseHandle(h2);
::Sleep(10000);                                             //程序睡眠 10 秒
return 0;
  }
DWORD WINAPI myfun1(LPVOID lpParameter)                     //线程函数 1
{
   m_Sec.Lock();                                            //锁定临界区
   a+=1;                                                    //变量加 1
   printf("%d",a);                                          //输出变量
```

```
   m_Sec.Unlock();                              //对临界区进行解锁
return 0;
    }
DWORD WINAPI myfun2(LPVOID lpParameter)         //线程函数 2
{
   m_Sec.Lock();                                //锁定临界区
   a+=1;                                        //变量加 1
   printf("%d",a);                              //输出变量
   m_Sec.Unlock();                              //对临界区进行解锁
return 0;
    }
```

用户使用 CCriticalSection 类或者临界区相关的 API 函数进行临界区编程，都可以使线程之间实现同步。在本小节中，主要讲解了怎样使用临界区对象实现线程同步以及与临界区编程相关的 API 函数和 MFC 类。

3.2.2　事件对象

事件对象是指用户在程序中使用内核对象的有无信号状态实现线程的同步。在本节中，将向用户介绍利用时间对象实现线程同步技术的相关 API 函数以及 MFC 类。

1．使用API函数操作事件对象

用户编程时，使用事件对象必须首先创建事件对象。在 API 函数中，用户可以使用函数 CreateEvent()创建并返回事件对象。该函数原型如下：

```
HANDLE CreateEvent(
  LPSECURITY_ATTRIBUTES lpEventAttributes,
  BOOL bManualReset,
  BOOL bInitialState,
  LPCTSTR lpName
);
```

如果该函数调用成功，则返回新创建的事件对象。其参数及意义如下：

- ❑ 参数 lpEventAttributes 是结构体 SECURITY_ATTRIBUTES 的指针，表示新创建的事件对象的安全属性。如果该参数为 NULL，则表示程序使用的是默认安全属性。
- ❑ 参数 bManualReset 表示所创建的事件对象是人工重置还是自动重置。如果该参数为 true，则表示程序所创建的事件对象为人工重置对象。如果为 flase，则表示创建的事件对象为自动重置对象。
- ❑ 参数 bInitialState 表示事件对象的初始状态。如果该参数为 true，则表示该事件对象初始时为有信号状态。否则，表示事件对象初始化时为无信号状态。
- ❑ 参数 lpName 表示事件对象的名称。如果该参数为 NULL，则表示程序创建的是一个匿名的事件对象。

注意：如果参数 bManualReset 设置为 true，则表示当调用线程获得其所有权后，用户需要显式地调用函数 ResetEvent()将时间对象设置为无信号状态。如果为自动重置的事件对象，则系统会自动将其设置为无信号状态。所以，一般情况下用户编程均将时间对象设置为自动重置。

例如，用户创建一个初始化状态为有信号并且是自动重置的事件对象。代码如下：

```
HANDLE hevent;                                    //定义事件对象
Hevent=::CreateEvent(NULL,false,true,NULL);       //创建事件对象
...                                               //省略部分代码
```

在程序中，用户创建了一个初始状态为有信号自动重置并且具有默认安全属性的匿名事件对象。

当用户创建时间对象时，如果将其初始状态设置为无信号，则需要用户手动将其设置为有信号状态。实现该功能可以调用函数 SetEvent()将指定的事件对象设置为有信号状态。该函数原型如下：

```
BOOL SetEvent(HANDLE hEvent);
```

该函数调用成功，则返回 true。否则，将返回 false。参数 hEvent 表示将设置的事件对象句柄。与该函数功能相反，函数 ResetEvent()则将指定的事件对象设置为无信号状态，其参数及意义与函数 SetEvent()相同。

当然，线程也可以通过调用函数 WaitForSingleObject()主动请求事件对象。该函数原型如下：

```
DWORD WaitForSingleObject(
  HANDLE hHandle,
  DWORD dwMilliseconds
);
```

该函数将在用户指定的事件对象上等待。如果事件对象处于有信号状态，函数将返回。否则，函数将一直等待，直到用户所指定的时间到达。各参数及其意义如下：

- ❑ 参数 hHandle 表示函数所等待的事件对象句柄。
- ❑ 参数 dwMilliseconds 表示该函数将在事件对象上的等待时间，如果该参数为 INFINITE，则该函数将永远等待。该函数的返回值可以表明引起函数返回的原因，其部分返回值如表 3.2 所示。

表 3.2　函数部分返回值

返　回　值	返回值意义
WAIT_TIMEOUT	用户指定的等待时间已过
WAIT_OBJECT_0	线程所请求的对象为有信号状态

例如，用户使用事件对象实现线程同步编程。代码如下：

```
#include <windows.h>                              //包含头文件
#include <stdio.h>
DWORD WINAPI myfun1(                              //声明线程函数
  LPVOID lpParameter
);
DWORD WINAPI myfun2(
  LPVOID lpParameter
);
HANDLE hevent;                                    //定义全局变量 hevent
int a=0;                                          //定义全局变量 a
main()
{
```

```
  HANDLE h1,h2;                                    //定义线程句柄
  hevent=::CreateEvent(NULL,FALSE,false,NULL);
  ::SetEvent(hevent);
 h1=::CreateThread(NULL,0,myfun1,NULL,0,NULL);    //创建线程 1
 printf("线程 1 开始运行！ \r\n");
 h2=::CreateThread(NULL,0,myfun2,NULL,0,NULL);    //创建线程 2
 printf("线程 2 开始运行！ \r\n");
 ::CloseHandle(h1);                               //关闭线程句柄对象
 ::CloseHandle(h2);
::Sleep(10000);                                   //程序睡眠 10 秒
return 0;
  }
DWORD WINAPI myfun1(LPVOID lpParameter)           //线程函数 1
{
   while(1)
   {
   ::WaitForSingleObject(hevent,INFINITE);        //请求事件对象
   ::ResetEvent(hevent);                          //设置事件对象为无信号状态
   if(a<10000)
{
 a+=1;                                            //变量自加
 ::Sleep(1000);                                   //线程睡眠 1 秒
   printf("线程 1: %d\r\n",a);                    //输出变量
     ::SetEvent(hevent);                          //设置事件对象为有信号状态
   }
else
{
    ::SetEvent(hevent);                           //设置事件对象为有信号状态
    break;                                        //跳出循环
   }
}
    return 0;
}
DWORD WINAPI myfun2(LPVOID lpParameter)           //线程函数 2
{
  while(1)
   {
   ::WaitForSingleObject(hevent,INFINITE);        //请求事件对象
   ::ResetEvent(hevent);                          //设置事件对象为无信号状态
   if(a<10000)
{
 a+=1;
 ::Sleep(1000);
   printf("线程 2: %d\r\n",a);                    //输出变量
     ::SetEvent(hevent);
  }
else
{
    ::SetEvent(hevent);                           //设置事件对象为有信号状态
    break;                                        //跳出循环
   }
}
    return 0;                                     //线程正常退出
    }
```

在代码中，用户主要使用了函数 WaitForSingleObject()对事件对象进行请求，然后再使用事件对象相关 API 函数设置其有无信号状态。实例程序根据事件对象的信号状态判断线

程的执行顺序以及输出全局变量 a 的值，如图 3.7 所示。

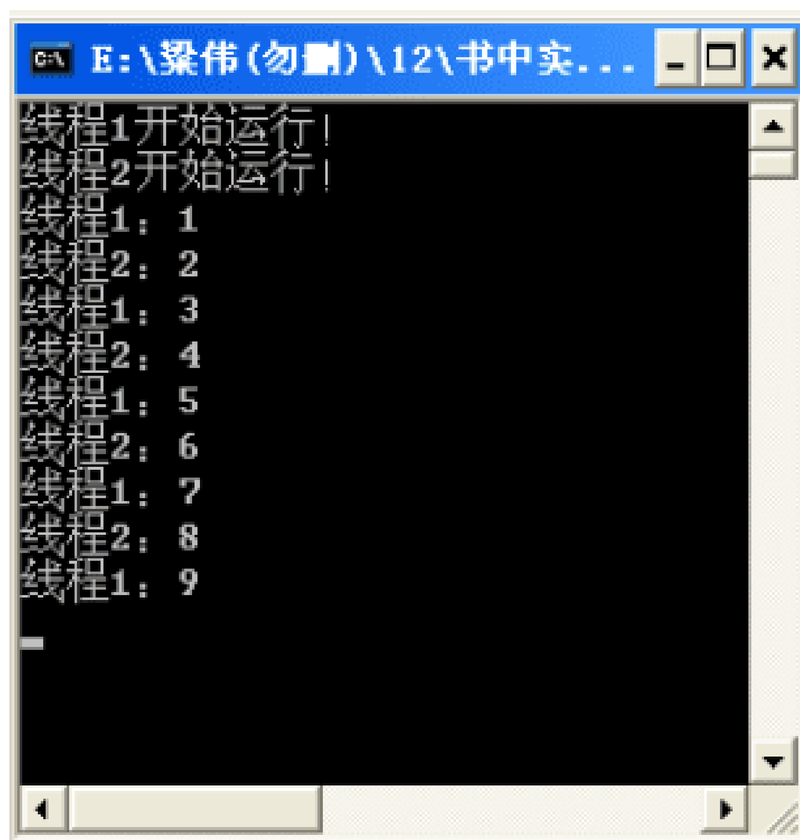


图 3.7　事件对象实现线程同步

在本小节中，向用户介绍了事件对象编程的相关 API 函数的原型以及参数意义，并配合实例程序讲解了这些 API 函数的使用方法。

2. 使用CEvent类实现线程同步

CEvent 类是 MFC 中支持事件对象编程的类。本小节将向用户讲解该类实现线程同步技术的部分常用函数以及使用方法。

首先，用调用该类构造函数创建对象。构造函数原型如下：

```
CEvent( BOOL bInitiallyOwn = FALSE, BOOL bManualReset = FALSE, LPCTSTR
lpszName = NULL, LPSECURITY_ATTRIBUTES lpsaAttribute = NULL );
```

在构造函数中，参数及其意义如下：

- 参数 bInitiallyOwn 表示事件对象的初始化状态。如果该参数为 true，则表示该事件对象为有信号状态。否则，该事件对象为无信号状态。默认为无信号状态。
- 参数 bManualReset 表示该事件对象是人工重置还是自动重置对象。如果该参数为 true，则事件对象为人工重置。否则，事件对象为自动重置。
- 参数 lpszName 表示用户为该事件对象的命名。默认情况下为 NULL。
- 参数 lpsaAttribute 表示该事件对象的安全属性。一般情况下，创建的事件对象均指定为默认安全属性。

例如，用户使用 CEvent 类创建对象。代码如下：

```
...                                           //省略部分代码
CEvent event(true,false,NULL,NULL);           //创建事件类对象
```

将事件对象设置为有信号或无信号状态可以分别调用函数 SetEvent()和 ResetEvent()。函数原型如下：

```
BOOL SetEvent( );                             //设置事件对象为有信号
BOOL ResetEvent( );                           //设置事件对象为无信号
```

以上两个函数若调用成功，则返回 true。否则，将返回 false。例如，用户使用 CEvent 类在程序中实现线程同步。代码如下：

```
#include <windows.h>                                      //包含头文件
#include <stdio.h>
DWORD WINAPI myfun1(                                      //声明线程函数
  LPVOID lpParameter
);
DWORD WINAPI myfun2(
  LPVOID lpParameter
);
CEvent event;                                             //将事件对象定义为全局变量
int a=0;                                                  //定义全局变量 a
main()
{
 event=CEvent(false,false,NULL,NULL);
  HANDLE h1,h2;                                           //定义线程句柄
  event=::CreateEvent(NULL,FALSE,false,NULL);
  event.SetEvent();
 h1=::CreateThread(NULL,0,myfun1,NULL,0,NULL); //创建线程 1
 printf("线程 1 开始运行！\r\n");
 h2=::CreateThread(NULL,0,myfun2,NULL,0,NULL); //创建线程 2
 printf("线程 2 开始运行！\r\n");
 ::CloseHandle(h1);                                       //关闭线程句柄对象
 ::CloseHandle(h2);
::Sleep(10000);                                           //程序睡眠 10 秒
return 0;
  }
DWORD WINAPI myfun1(LPVOID lpParameter)                   //线程函数 1
{
   while(1)
   {
   ::WaitForSingleObject(event.m_hObject,INFINITE);       //请求事件对象
  event.ResetEvent();                                     //设置事件对象为无信号状态
   if(a<10000)
{
 a+=1;                                                    //变量自加
 ::Sleep(1000);                                           //线程睡眠 1 秒
   printf("线程 1: %d\r\n",a);                            //输出变量
    event.SetEvent();                                     //设置事件对象为有信号状态
   }
else
{
  event.SetEvent();                                       //设置事件对象为有信号状态
    break;                                                //跳出循环
   }
}
    return 0;                                             //线程
}
DWORD WINAPI myfun2(LPVOID lpParameter)                   //线程函数 2
{
  while(1)
   {
   ::WaitForSingleObject(event.m_hObject,INFINITE);       //请求事件对象
   event.ResetEvent();                                    //设置事件对象为无信号状态
   if(a<10000)
{
 a+=1;
 ::Sleep(1000);
   printf("线程 2: %d\r\n",a);                            //输出变量
```

```
     event.SetEvent();
   }
else
{
    event.SetEvent();                                  //设置事件对象为有信号状态
    break;                                             //跳出循环
   }
}
    return 0;                                          //线程正常退出
    }
```

在上面的程序中，用户主要使用了 CEvent 类的相关函数实现线程的同步。本小节主要向用户讲解如何使用 API 函数或者 CEvent 类创建事件对象，实现线程同步技术的相关方法。

3.2.3　互斥对象

互斥对象与前面所学的临界区对象和事件对象的作用一样，均用于实现线程同步。但是，互斥对象还可以在进程之间使用。在互斥对象中，包含一个线程 ID 和一个计数器。线程 ID 表示拥有该互斥对象的线程，计数器用于表示该互斥对象被同一线程所使有的次数。在程序中，同样可以使用 API 函数或者 MFC 类操作互斥对象，实现线程同步。

1．使用API函数操作互斥对象

用户可以调用 API 函数 CreateMutex()创建并返回互斥对象。该函数原型如下：

```
HANDLE CreateMutex(
  LPSECURITY_ATTRIBUTES lpMutexAttributes,
   BOOL bInitialOwner,
  LPCTSTR lpName
);
```

如果该函数调用成功，将返回新创建的互斥对象句柄。否则，将返回 NULL。各参数及其意义如下：

- 参数 lpMutexAttributes 指定新创建互斥对象的安全属性。如果该参数为 NULL，表示互斥对象拥有默认的安全属性。
- 参数 bInitialOwner 表示该互斥对象的拥有者。如果为 true，则表示创建该互斥对象的线程拥有其所有权。如果为 false，表示创建互斥对象的线程不能拥有该互斥对象的所有权。
- 参数 lpName 表示互斥对象的名称。若该参数为 NULL，则表示程序创建的是匿名对象。如果用户为该参数指定值，则在程序中可以调用函数 OpenMutex()打开一个命名的互斥对象。

例如，用户创建一个匿名的互斥对象，代码如下：

```
HANDLE hmutex;                                  //声明互斥对象句柄
hmutex=::CreateMutex(NULL,FALSE,NULL):          //创建互斥对象并返回其句柄
...                                             //省略部分代码
```

线程使用完该互斥对象以后，用户应该调用函数 ReleaseMutex()释放对该互斥对象的

所有权，也就是让互斥对象处于有信号状态。函数 ReleaseMutex()的原型如下：

```
BOOL ReleaseMutex(HANDLE hMutex);
```

如果该函数调用成功，则返回 true。否则，将返回 false。参数 hMutex 表示将释放的互斥对象句柄。例如，用户将上面创建的互斥对象句柄 hmutex 与调用该句柄的线程进行分离。代码如下：

```
...                                                  //省略部分代码
::ReleaseMutex(hmutex);                              //释放互斥对象句柄
```

在互斥对象中，线程也可以调用函数 WaitForSingleObject()对该对象进行请求。当互斥对象无信号时，该函数将一直等待，直到该互斥对象有信号或用户所指定的等待时间已过。否则，该函数将返回。由于这个函数在 3.2.2 节中已经详细讲解，所以在这里不再赘述。如果用户对该函数不了解，请复习前面的相关知识。

例如，用户使用互斥对象实现线程的同步，代码如下：

```
#include <windows.h>                                //包含头文件
#include <stdio.h>
DWORD WINAPI myfun1(                                //声明线程函数
  LPVOID lpParameter
);
DWORD WINAPI myfun2(
  LPVOID lpParameter
);
HANDLE hmutex;
int a=0;                                            //定义全局变量 a
main()
{
hmutex=::CreateMutex(NULL,FALSE,NULL);              //创建互斥对象并返回其句柄
  HANDLE h1,h2;                                     //定义线程句柄
 h1=::CreateThread(NULL,0,myfun1,NULL,0,NULL);      //创建线程 1
 printf("线程 1 开始运行！\r\n");
 h2=::CreateThread(NULL,0,myfun2,NULL,0,NULL);      //创建线程 2
 printf("线程 2 开始运行！\r\n");
 ::CloseHandle(h1);                                 //关闭线程句柄对象
 ::CloseHandle(h2);
::Sleep(10000);                                     //程序睡眠 10 秒
return 0;
  }
DWORD WINAPI myfun1(LPVOID lpParameter)             //线程函数 1
{
   while(1)
   {
   ::WaitForSingleObject(hmutex,INFINITE);          //请求互斥对象
   if(a<10000)
{
 a+=1;                                              //变量自加
 ::Sleep(1000);                                     //线程睡眠 1 秒
   printf("线程 1: %d\r\n",a);
  ::ReleaseMutex(hmutex);                           //释放互斥对象句柄
   }
else
{
::ReleaseMutex(hmutex);                             //释放互斥对象句柄
```

```
    break;                                  //跳出循环
   }
}
    return 0;
}
DWORD WINAPI myfun2(LPVOID lpParameter)     //线程函数 2
{
  while(1)
   {
   ::WaitForSingleObject(hmutex,INFINITE);  //请求互斥对象
if(a<10000)
 {
  a+=1;
  ::Sleep(1000);
   printf("线程 2: %d\r\n",a);              //输出变量
   ::ReleaseMutex(hmutex);                  //释放互斥对象句柄
  }
else
{
 ::ReleaseMutex(hmutex);                    //释放互斥对象句柄
    break;                                  //跳出循环
   }
}
    return 0;                               //线程正常退出
    }
```

在程序中，用户首先创建互斥对象并返回其句柄。然后利用该互斥对象句柄实现线程 1 和线程 2 的同步并输出全局变量 a 的值，如图 3.8 所示。

注意：当用户使用互斥对象编程时，应该牢记互斥对象的使用方法是哪个线程拥有其所有权，哪个线程就应该释放该互斥对象。

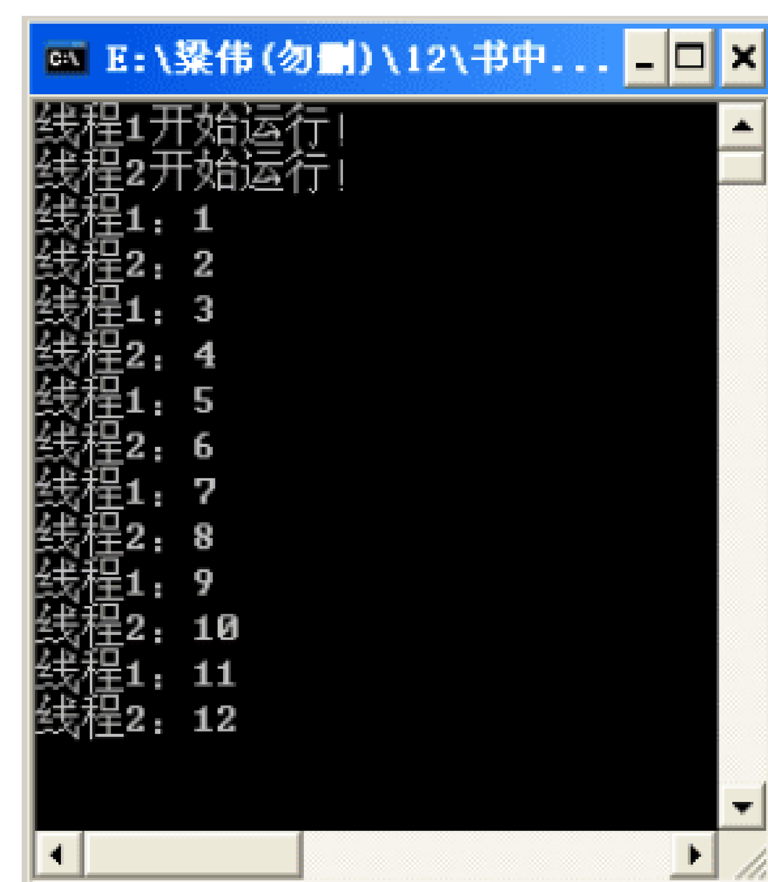


图 3.8　使用互斥对象实现线程同步

2. 使用CMutex类

CMutex 类是 MFC 中的互斥对象类。该类是由 CSyncObject 类派生而来，所以使用 CMutex 类时可以调用其父类 CSyncObject 中的成员函数实现指定功能。在本小节中，将向用户讲解使用 CMutex 类实现线程同步技术的方法。

首先，创建 CMutex 类对象是通过其构造函数实现的。构造函数原型如下：

```
CMutex(BOOL bInitiallyOwn=FALSE,LPCTSTR lpszName=NULL,LPSECURITY_ ATTRIB-
UTES lpsaAttribute = NULL );
```

该函数的作用是构造 CMutex 类的实例对象。其参数及意义如下：

- 参数 bInitiallyOwn 表示调用线程是否拥有所创建的互斥对象。如果为 true，则表示创建该互斥对象的线程拥有其所有权。如果为 false，表示创建互斥对象的线程不能拥有该互斥对象的所有权。
- 参数 lpName 表示互斥对象的名称。若该参数为 NULL，则表示程序创建的是匿名对象。

- 参数 lpMutexAttributes 指定新创建互斥对象的安全属性。如果该参数为 NULL，表示互斥对象拥有默认的安全属性。

例如，用户通过 CMutex 类创建一个互斥对象。代码如下：

```
...                                         //省略部分代码
CMutex mex(FALSE,NULL,NULL);                //创建互斥对象
...                                         //省略部分代码
```

用户创建互斥对象成功之后，可以在线程函数中调用函数 Lock()和 Unlock()对该互斥对象所保护的区域进行锁定和解锁，控制其他线程对保护区域的访问权限。这两个函数的原型如下：

```
virtual BOOL Lock( DWORD dwTimeout = INFINITE);
virtualBOOL Unlock(LONG lCount,LPLONG lpPrevCount=NULL);
```

其中，函数 Lock()的作用是锁定保护区域的数据，避免其他线程对该区域数据进行访问，并且将互斥对象设置为无信号。如果该函数调用成功，则返回 true，否则返回 false。其参数 dwTimeout 表示该互斥对象变为有信号状态的时间。如果为 INFINITE，则表示该函数将一直等待，直到互斥对象变为有信号状态。

函数 Unlock()的作用是解除对保护区域数据的锁定，并将互斥对象设置为有信号状态。如果该函数调用成功，则返回 true，否则返回 false。其参数 lCount 是默认参数，用户在使用时可以不为其指定值。参数 lpPrevCount 也是默认参数，默认为 NULL。

例如，用户在程序中使用 CMutex 类创建互斥对象实现线程同步。代码如下：

```
#include <windows.h>                               //包含头文件
#include <stdio.h>
DWORD WINAPI myfun1(                               //声明线程函数
  LPVOID lpParameter
);
DWORD WINAPI myfun2(
  LPVOID lpParameter
);
CMutex hmutex(NULL,FALSE,NULL);                    //定义全局互斥对象
int a=0;                                           //定义全局变量 a
main()
{
  HANDLE h1,h2;                                    //定义线程句柄
 h1=::CreateThread(NULL,0,myfun1,NULL,0,NULL);     //创建线程 1
printf("使用 CMutex 类实现线程同步\r\n");
 printf("线程 1 开始运行！\r\n");
 h2=::CreateThread(NULL,0,myfun2,NULL,0,NULL);     //创建线程 2
 printf("线程 2 开始运行！\r\n");
 ::CloseHandle(h1);                                //关闭线程句柄对象
 ::CloseHandle(h2);
::Sleep(10000);                                    //程序睡眠 10 秒
return 0;
  }
DWORD WINAPI myfun1(LPVOID lpParameter)            //线程函数 1
{
   while(1)
   {
   hmutex.Lock(INFINITE);                          //锁定互斥对象
   if(a<10000)
```

```
{
 a+=1;                                       //变量自加
 ::Sleep(1000);                              //线程睡眠 1 秒
   printf("线程 1: %d\r\n",a);
  hmutex.Unlock(0,0);                        //释放互斥对象
   }
else
{
hmutex.Unlock(0,0);                          //释放互斥对象
    break;                                   //跳出循环
   }
}
    return 0;
}
DWORD WINAPI myfun2(LPVOID lpParameter)      //线程函数 2
{
  while(1)
   {
   hmutex.Lock(INFINITE);                    //锁定互斥对象
if(a<10000)
{
 a+=1;
 ::Sleep(1000);
   printf("线程 2: %d\r\n",a);               //输出变量
   hmutex.Unlock(0,0);                       //释放互斥对象
  }
else
{
 hmutex.Unlock(0,0);                         /释放互斥对象
    break;                                   //跳出循环
   }
}
    return 0;                                //线程正常退出
    }
```

在程序中，用户首先定义了 CMutex 类的全局对象，然后创建两个线程并在线程函数中调用函数 Lock()和 Unlock()实现线程同步。该程序的运行结果如图 3.9 所示。

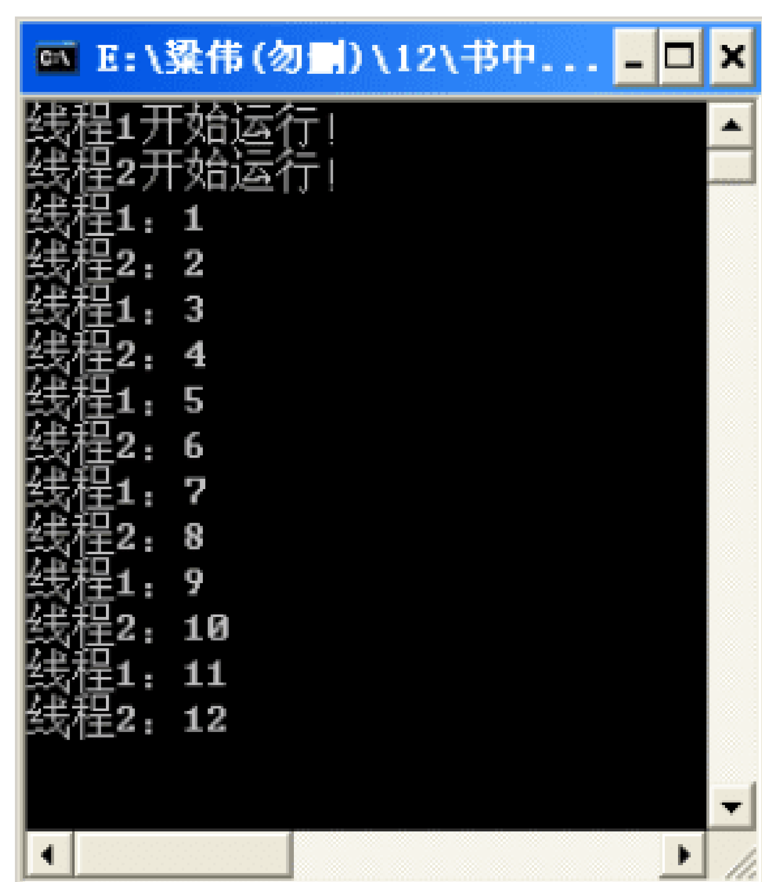


图 3.9　使用 CMutex 类实现线程同步

以上程序实现了在同一进程中的线程同步，但是在前面的内容中向用户介绍过互斥对象还可以在进程之间使用。如果用户在进程中通过创建互斥对象实现程序实例的唯一运行，代码如下：

```
#include<windows.h>                                    //包含头文件
#include<stdio.h>
main()                                                 //主函数
{
 HANDLE hmutex;                                        /定义互斥对象句柄
 hmutex=::CreateMutex(NULL,true,"VC 网络编程");        /创建互斥对象并返回其句柄
  if(hmutex)                                           //判断创建互斥对象是否成功
 {
 if(ERROR_ALREADY_EXISTS==GetLastError())              //获取错误
  {
      printf("只允许一个实例程序运行！ \r\n");         //打印相关信息
     }
  else
  {
     printf("实例程序运行成功！ \r\n");
    }
}
 ::ReleaseMutex(hmutex);                               //释放互斥对象句柄
  ::Sleep(100000);                                     //使程序睡眠 100 秒
   return 0;                                           //程序正常结束
}
```

在代码中，用户首先创建互斥对象，然后使用函数 GetLastError()获取错误信息。如果获取到的错误信息是 ERROR_ALREADY_EXISTS，则说明程序已经有一个实例在运行了。该程序的运行结果如图 3.10 所示。

在本小节中，主要向用户介绍了使用 API 函数和 CMutex 类创建互斥对象以及使用互斥对象实现线程同步的方法。

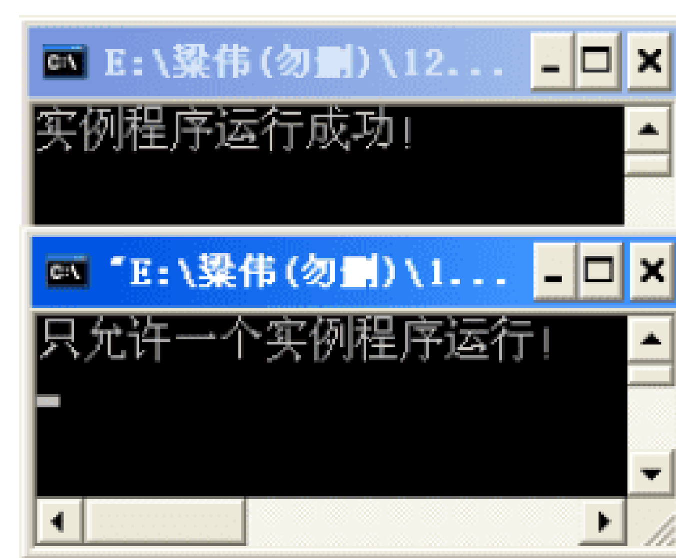


图 3.10　实例程序运行唯一性

3.3　进程间通信

进程间通信是指在系统中两个或多个进程之间通过第三方进行数据共享。用户在实际编程中，除了可以使用套接字进行网络通信以外，还可以使用进程间的通信方式实现网络通信。例如，邮槽、命名管道等。

在 Windows 操作系统中，当每个进程启动时，系统都会为其分配大约 4GB 的私有地址空间。由于每个进程的地址空间是私有的，所以进程之间不能互相访问对方的数据。但是，在 Windows 操作系统中已经为用户提供了多种进程通信机制。例如，邮槽、匿名管道等。在本节中，将主要向用户介绍这些通信机制的用法以及实现方法等。

3.3.1　邮槽

邮槽是 Windows 系统提供的一种单向通信的机制。即进程中的一方只能写入或读取数据，而另一方则只能读取或写入数据。通过邮槽，用户可以实现一对多或跨网络的进程之间的通信。但是，邮槽能传输的数据非常小，一般在 400KB 左右。如果用户操作的数据过

大，可能会导致邮槽不能正常工作。

1．创建邮槽

用户在实际编程时，可以使用 Windows 邮槽实现进程间通信。但是，用户必须首先创建邮槽。在 Windows 操作系统中，用户可以通过函数 CreateMailslot()创建邮槽。该函数原型如下：

```
HANDLE CreateMailslot(
  LPCTSTR lpName,
  DWORD nMaxMessageSize,
  DWORD lReadTimeout,
  LPSECURITY ATTRIBUTES lpSecurityAttributes
);
```

该函数的作用是创建邮槽并返回该邮槽的句柄。如果该函数调用成功，将返回创建邮槽的句柄。否则，函数将返回 INVALID_HANDLE_VALUE，表示创建邮槽失败。其参数及意义如下：

- ❑ 参数 lpName 表示邮槽的名称。邮槽名称的格式为"\\.\mailslot\name"。其中，name 表示邮槽的名称。用户在 VC 中使用该参数时，应该将其指定为"\\\\.\\mailslot\\name"。如果用户是在不同的主机上运行程序，则需要将名称字符串中的"."换成对方主机名称。
- ❑ 参数 nMaxMessageSize 指定将通过邮槽发送或接收的消息大小的最大值。用户在实际编程时，一般将该参数设置为 0，表示消息的大小为任意值。
- ❑ 参数 lReadTimeout 表示程序读取操作的超时时间。如果该参数值为 0，则当邮槽中没有任何消息时，该函数将立即返回。如果该参数值为 MAILSLOT_WAIT_FOREVER，则表示该函数将等待，直到邮槽中有消息，函数才会返回。
- ❑ 参数 lpSecurityAttributes 是指向结构体 SECURITY_ATTRIBUTES 的指针，表示邮槽的安全属性。一般情况下，用户将该参数值指定为 NULL，表示邮槽使用默认的安全属性。

一般情况下，函数 CreateMailslot()常被使用在进程通信的服务器方。在客户端则使用函数 CreateFile()打开指定的邮槽之后，再进行相关的操作。

注意：*在本节中，将通过邮槽读取数据的通信一方称为服务器，而通过邮槽写入数据的一方称为客户端。*

例如，用户分别在服务器和客户端上创建邮槽和打开邮槽。服务器端创建邮槽的代码如下：

```
...                                                          //省略部分代码
HANDLE mail;                                                 //定义邮槽句柄
mail=CreateMailslot("\\\\.\\mailslot\\mysolt",0,  MAILSLOT_WAIT_FOREVER,
ULL);                                                        //创建邮槽
if(mail==INVALID HANDLE VALUE)                               //判断邮槽句柄
{
  MessageBox("创建邮槽失败！");                              //提示信息
 }
...                                                          //省略部分代码
```

```
//客户端打开邮槽
...                                                         //省略部分代码
HANDLE mail2;                                               //定义文件句柄
mail2=CreateFile("\\\\.\\mailslot\\mysolt",  GENERIC WRITE,  FILE SHARE
READ,NULL, OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL,NULL);      //打开文件
if(mail2== INVALID_HANDLE_VALUE)
{
  MessageBox("打开邮槽失败！");
  }
...                                                         //省略部分代码
```

用户首先在服务器方创建邮槽“\\\\.\\mailslot\\mysolt”。然后，再在客户端创建并打开与该邮槽相关联的文件。

注意： 如果用户需要在程序中既能读取数据又能写入数据，则只需要在程序中同时实现服务器与客户端的功能即可。

2．操作邮槽

用户对邮槽进行操作包括将数据写入邮槽和从邮槽中读取数据等。在实际编程时，用户操作邮槽与操作文件一样，都是通过调用函数 ReadFile()和 WriteFile()进行读写操作。例如，用户在服务器方通过邮槽读取数据，而在客户端通过邮槽写入数据。代码如下：

```
//服务器方读取数据
...                                                     //省略部分代码
char text[200];                                         //定义字符数组
DWORD readtext;                                         //用于获取实际读取值
if(ReadFile(mail,text,200,&readtext,NULL))              //读取数据
{
  MessageBox(text);                                     //显示数据
   }
else
{
  MessageBox("数据读取失败!");
 }
...                                                     //省略部分代码
//客户端写入数据
...                                                     //省略部分代码
char text[]="this is a message";                        //初始化消息
DWORD writetext;                                        //用于获取实际发送值
if(WriteFile(mail2,text,sizeof(text),&writetext,NULL))  //写入数据
{
  MessageBox("数据写入成功");                           //数据写入成功
  }
else
{
  MessageBox("数据写入失败");
  }
...                                                     //省略部分代码
```

在以上代码中，用户实现了简单的邮槽操作。用户在程序中使用完邮槽之后，必须调用函数 CloseHandle()将创建的邮槽关闭。

3．邮槽实例

首先，在 VC 中创建一个基于控制台程序的窗口工程，名称为“邮槽实例”。然后，在该工程中添加一个 C++源文件，名称修改为“服务器”并添加代码。邮槽服务器代码如下：

```
#include<windows.h>                                              //包含头文件
#include<stdio.h>
main()                                                           //主函数
{
HANDLE mail;                                                     //定义邮槽句柄
mail=CreateMailslot("\\\\.\\mailslot\\mysolt",0,MAILSLOT_WAIT_FOREVER,
NULL);                                                           //创建邮槽
if(mail==INVALID_HANDLE_VALUE)                                   //判断邮槽句柄
{
 printf("创建邮槽失败！ \r\n");                                   //提示信息
 CloseHandle(mail);
  }
else
{
printf("创建邮槽成功，正在读取数据……！ \r\n");
char text[200];                                                  //定义字符数组
DWORD readtext;                                                  //获取实际读取值
if(ReadFile(mail,text,200,&readtext,NULL))                       //读取数据
{
  printf(text);                                                  //显示数据
   }
else
{
  printf("\r\n 数据读取失败!\r\n");
 }}
CloseHandle(mail);                                               //关闭邮槽
Sleep(1000);
return 0;
}
```

将上面的代码编译后生成邮槽服务器程序。再在工程中添加一个 C++源文件，名称修改为“客户端”并添加代码。客户端代码如下：

```
#include<windows.h>                                              //包含头文件
#include<stdio.h>
main()                                                           //主函数
{
HANDLE mail2;                                                    //定义邮槽句柄
char text[]="您好，this is a message";                           //初始化消息
DWORD writetext;                                                 //获取实际发送值
mail2=CreateFile("\\\\.\\mailslot\\my",GENERIC_WRITE,FILE_SHARE_READ,
               NULL, OPEN_EXISTING,FILE_ATTRIBUTE_NORMAL,NULL);
                                                                 //打开文件
if(INVALID_HANDLE_VALUE==mail2)
{
  printf("邮槽打开失败！ \r\n");
  }
else
{
```

```
 if(WriteFile(mail2,text,sizeof(text),&writetext,NULL))    //写入数据
{
  Sleep(1000);
    printf("数据写入成功\r\n");                              //数据写入成功
  }
else
{
  printf("数据写入失败\r\n");
  }
CloseHandle(mail2);                                          //关闭句柄
}
Sleep(10000);
return 0;
}
```

当用户实现了邮槽服务器和邮槽客户端的相关功能，便可以编译并运行邮槽服务器和邮槽客户端，如图 3.11 所示。

注意：在邮槽实例中，用户必须首先打开服务器程序创建邮槽。然后，再使用客户端打开邮槽写入数据。如果用户将两个程序打开的顺序弄反，则会导致程序功能发生错误。

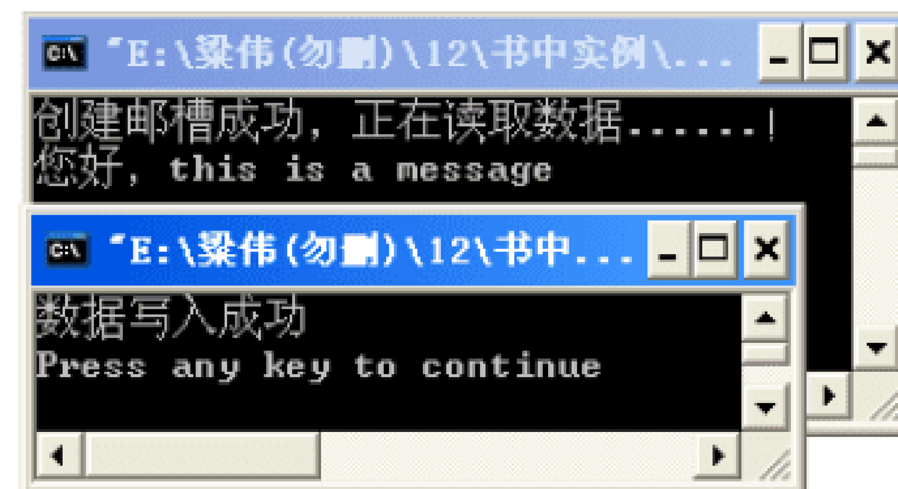


图 3.11　邮槽实例程序运行结果

3.3.2　命名管道

命名管道是一种不但能在同一机器上实现两个进程通信，还能在网络中不同机器上的两个进程之间通信的机制。与邮槽不同，命名管道传输数据是采取基于连接并且可靠的传输方式，所以命名管道传输数据只能一对一进行传输。在本节中，将主要向用户介绍命名管道的使用方法。

1．创建命名管道

用户创建命名管道可以调用函数 CreateNamedPipe()进行创建。该函数原型如下：

```
HANDLE CreateNamedPipe(
  LPCTSTR lpName,
  DWORD dwOpenMode,
  DWORD dwPipeMode,
  DWORD nMaxInstances,
  DWORD nOutBufferSize,
  DWORD nInBufferSize,
  DWORD nDefaultTimeOut,
  LPSECURITY_ATTRIBUTES lpSecurityAttributes
);
```

如果该函数调用成功，则返回创建的命名管道句柄。否则，该函数返回 INVALID_HANDLE_VALUE。各参数及其意义如下：

- 参数 lpName 表示创建的命名管道名称。该名称格式为“\\.\pipe\pipename”。但是，用户在实际编程时，应该将该名称修改为“\\\\.\\pipe\\pipename”。如果用户希望

在不同计算机的两个进程之间进行通信，则需要将名称字符串中的符号“.”修改为远程计算机的名称即可。

- 参数 dwOpenMode 表示名管道的打开模式，包括访问模式、管道句柄的安全访问模式以及重叠方式等。该参数取值，如表 3.3 所示。

表 3.3　命名管道打开模式取值

模 式 取 值	意　　义
PIPE_ACCESS_DUPLEX	指定双向模式，即服务器与客户端都可以从命名管道中读取或写入数据
PIPE_ACCESS_INBOUND	命名管道的数据只能从客户端到服务器，即用户指定该模式表示服务器只能读取数据而客户端只能写入数据
PIPE_ACCESS_OUTBOUND	命名管道的数据只能从服务器到客户端，即用户指定该模式表示服务器只能写入数据而客户端只能读取数据
FILE_FLAG_WRITE_THROUGH	允许写直通模式。当用户指定该值时，写入数据的一方要等到写入的数据到达另一方的数据缓冲区之后，才会成功返回
FILE_FLAG_OVERLAPPED	允许使用重叠模式。采用该模式可以使一些耗费时间的操作在后台执行，在重叠模式下，一个线程可以在多个管道实例上同时处理输入与输出操作
WRITE_DAC	调用线程对命名管道的任意访问控制列表都可以进行写入操作
WRITE_OWNER	调用者对命名管道的所有者可以进行写入操作
ACCESS_SYSTEM_SECURITY	调用者对命名管道的安全访问控制列表可以进行写入操作

- 参数 dwPipeMode 表示句柄管道的类型、读取以及等待方式。该参数的具体取值，如表 3.4 所示。

表 3.4　管道句柄、读取以及等待方式

取　　值	意　　义
PIPE_TYPE_BYTE	数据以字节流的形式写入管道
PIPE_TYPE_MESSAGE	数据以消息流的形式写入管道
PIPE_READMODE_BYTE	以字节流的形式从管道中读取数据
PIPE_READMODE_MESSAGE	以消息流的形式从管道中读取数据
PIPE_WAIT	允许阻塞模式
PIPE_NOWAIT	允许非阻塞方式

- 参数 nMaxInstances 表示管道能够创建实例的最大数目。其取值范围在 1～PIPE_UNLIMITED_INSTANCES。如果将该值设为 PIPE_UNLIMITED_INSTANCES，则创建的管道实例数目仅限于操作系统。

注意：一个客户端只能与一个管道实例进行通信。

- 参数 nOutBufferSize 表示输出缓冲区的大小。
- 参数 nInBufferSize 表示输入缓冲区的大小。
- 参数 nDefaultTimeOut 表示超时值，使用同一管道的不同实例必须将该参数取同样的超时值。
- 参数 lpSecurityAttributes 是指向结构体 SECURITY_ATTRIBUTES 的指针，表示命

名管道的安全属性。

例如，用户使用该函数创建一个命名管道。代码如下：

```
...                                              //省略部分代码
HANDLE hpip;
hpip=CreateNamedPipe("\\\\.\\pipe\\pipename", PIPE_ACCESS_DUPLEX,
                        PIPE_TYPE_BYTE, PIPE_UNLIMITED_INSTANCES,1024,1024,0,
                        NULL);
                                                 //创建命名管道
...                                              //省略部分代码
```

2. 连接命名管道

当用户成功创建命名管道以后，便可以调用相关函数连接该命名管道。但是，服务器与客户端连接命名管道的方法并不一样。

对于服务器而言，可以调用函数 ConnectNamedPipe()等待客户端的连接请求。该函数原型如下：

```
BOOL ConnectNamedPipe(
  HANDLE hNamedPipe,
  LPOVERLAPPED lpOverlapped
);
```

该函数只能对命名管道的服务器方进行调用，其作用是等待客户端的连接请求。其参数 hNamedPipe 表示命名管道的句柄。参数 lpOverlapped 是指向结构体 OVERLAPPED 的指针，如果创建的管道是使用 FILE_FLAG_OVERLAPPED 标记打开，那么该参数指向的结构体中必须包含一个人工重置的事件对象。例如，用户在服务器端使用该函数等待客户端的连接请求，代码如下：

```
...                                                      //省略部分代码
OVERLAPPED ovi={0};                                      //定义结构体变量
if(::ConnectNamedPipe(hpip,&ovi))                        //等待客户端的连接请求

{
      MessageBox("客户端连接成功！");                     //提示信息
}
...                                                      //省略部分代码
```

对于通信的客户端而言，需要在连接服务器创建的命名管道之前判断该命名管道是否可用。用户在程序中实现这个功能可以调用函数 WaitNamedPipe()。该函数原型如下：

```
BOOL WaitNamedPipe(
  LPCTSTR lpNamedPipeName,
  DWORD nTimeOut
);
```

该函数的作用是判断服务器创建的命名管道是否可用。其参数及意义如下：

- 参数 lpNamedPipeName 表示命名管道的名称。该名称的格式也是“\\\\.\\pipe\\pipename”。如果用户希望在不同计算机的两个进程之间进行通信，则需要将名称字符串中的符号“.”修改为远程计算机的名称。
- 参数 nTimeOut 表示超时的时间间隔。其取值如表 3.5 所示。

表 3.5　参数取值

取　　值	意　　义
NMPWAIT_USE_DEFAULT_WAIT	表示超时时间是服务器创建命名管道时所指定的超时时间
NMPWAIT_WAIT_FOREVER	表示该函数将一直等待，直到出现可用的命名管道

如果该函数调用成功，将返回 true。否则，函数将返回 false。当函数 WaitNamedPipe()调用成功后，用户需要使用函数 CreateFile()将该命名管道打开以获得该管道的句柄。例如，用户在客户端获取服务器创建的命名管道句柄。代码如下：

```
...                                                       //省略部分代码
HANDLE hpip;
if(WaitNamedPipe("\\\\.\\pipe\\pipename", NMPWAIT_WAIT_FOREVER))
                                                          //连接命名管道
{
  hpip=CreateFile("\\\\.\\pipe\\pipename", GENERIC_READ| GENERIC_WRITE,0,
           NULL, OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL,NULL);
                                                          //打开指定命名管道
  }
else
{
   MessageBox("连接命名管道失败");                          //提示信息
  }
```

在本小节中，分别向用户介绍了服务器与客户端连接命名管道的方法。

3．读写命名管道

不论服务器还是客户端，只要双方的命名管道连接成功，用户便可以调用函数 ReadFile()和 WriteFile()对命名管道进行读写操作。例如，用户通过命名管道读取数据。代码如下：

```
...                                                 //省略部分代码
char buf[200];                                      //定义数据缓冲区
DWORD readbuf;                                      //获取实际读取字节数
if(ReadFile(hpip,buf,200,&readbuf,NULL))            //读取管道数据
{
  MessageBox("数据读取成功");                        //提示信息
  }
else
{
  MessageBox("数据读取失败");
  }
...                                                 //省略部分代码
```

以上代码的作用是服务器或客户端通过函数 ReadFile()读取命名管道中的数据。如果读取数据成功，则提示用户数据读取成功。

如果用户需要写入数据到命名管道中，可以调用函数 WriteFile()进行数据写入。代码如下：

```
...                                                 //省略部分代码
char buf[]="测试程序";                               //定义数据缓冲区
DWORD readbuf;                                      //获取实际读取字节数
if(WriteFile(hpip,buf,sizeof(buf),&readbuf,NULL))   //写入数据到管道
```

```
{
  MessageBox("数据写入成功");                          //提示信息
  }
else
{
  MessageBox("数据写入失败");
  }
...                                                    //省略部分代码
```

用户使用完命名管道之后，必须调用函数 CloseHandle()将命名管道的句柄删除。代码如下：

```
...                                                    //省略部分代码
CloseHandle(hpip);                                     //关闭命名管道句柄
```

通过以上代码，用户已经可以在实例程序中使用命名管道传输数据了。

4．命名管道实例

在本小节中，将通过命名管道实例程序向用户讲解命名管道的具体使用方法。在 VC 中创建基于控制台的工程，并将工程名修改为“命名管道实例”。然后添加一个 C++源文件，名称为“服务器”，添加代码如下：

```
#include<windows.h>                                    //包含头文件
#include<stdio.h>
main()
{
 HANDLE hpip;                                          //定义命名管道句柄
 OVERLAPPED ovi={0};                                   //定义结构体变量
  char buf[200];                                       //定义数据缓冲区
 DWORD readbuf;                                        //获取实际读取字节数
 hpip=CreateNamedPipe("\\\\.\\pipe\\pipename", PIPE_ACCESS_DUPLEX,
                    PIPE_TYPE_BYTE, PIPE_UNLIMITED_INSTANCES,1024,1024,0,
                    NULL);                             //创建命名管道
printf("创建管道成功，正在等待客户端连接！\r\n");
if(::ConnectNamedPipe(hpip,&ovi))                      //等待客户端的连接请求
{
  printf("客户端连接成功！\r\n");
  printf("正在读取数据！\r\n");                         //提示信息
if(ReadFile(hpip,buf,200,&readbuf,NULL))               //读取管道数据
{
  printf("数据读取成功\r\n");                           //提示信息
  printf("读取的数据是：%s\r\n",buf);
  }
else
{
  printf("数据读取失败\r\n");
  }
}}
```

将上面的代码编译之后，会生成命名管道服务器。然后在工程中添加一个 C++源文件，名称修改为“客户端”，添加代码如下：

```
#include<windows.h>                                    //包含头文件
#include<stdio.h>
main()
```

```
{
 HANDLE hpip;
 OVERLAPPED ovi={0};
  char buf[]="命名管道测试程序";                              //定义数据缓冲区
 DWORD readbuf;                                              //定义结构体变量
 printf("正在连接命名管道! \r\n");
 if(WaitNamedPipe("\\\\.\\pipe\\pipename", NMPWAIT_WAIT_FOREVER))
                                                             //连接命名管道
{
  hpip=CreateFile("\\\\.\\pipe\\pipename", GENERIC_READ| GENERIC_WRITE,0,
           NULL, OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL,NULL);
                                                             //打开指定命名管道
   if(hpip==INVALID_HANDLE_VALUE)                            //打开命名管道失败
   {
    printf("打开命名管道失败\r\n");
   }
   else
   {
if(WriteFile(hpip,buf,sizeof(buf),&readbuf,NULL))            //写入数据到管道
{
  printf("数据写入成功\r\n");                                //提示信息
  }
else
{
  printf("数据写入失败\r\n");
  }
   }
 }
else
{
   printf("连接命名管道失败\r\n");                           //提示信息
  }                                                          //创建命名管
}
```

用户将客户端代码编译之后，将前面已经编译好的服务器程序打开。用户可以看到服务器与客户端如何通过命名管道传输数据，如图 3.12 所示。

通过本节的学习，用户可以非常熟练地使用命名管道在两个进程之间进行数据传输。用户还可以将书中的实例程序与随书光盘中的实例程序进行对比学习，会使学习效果更好。

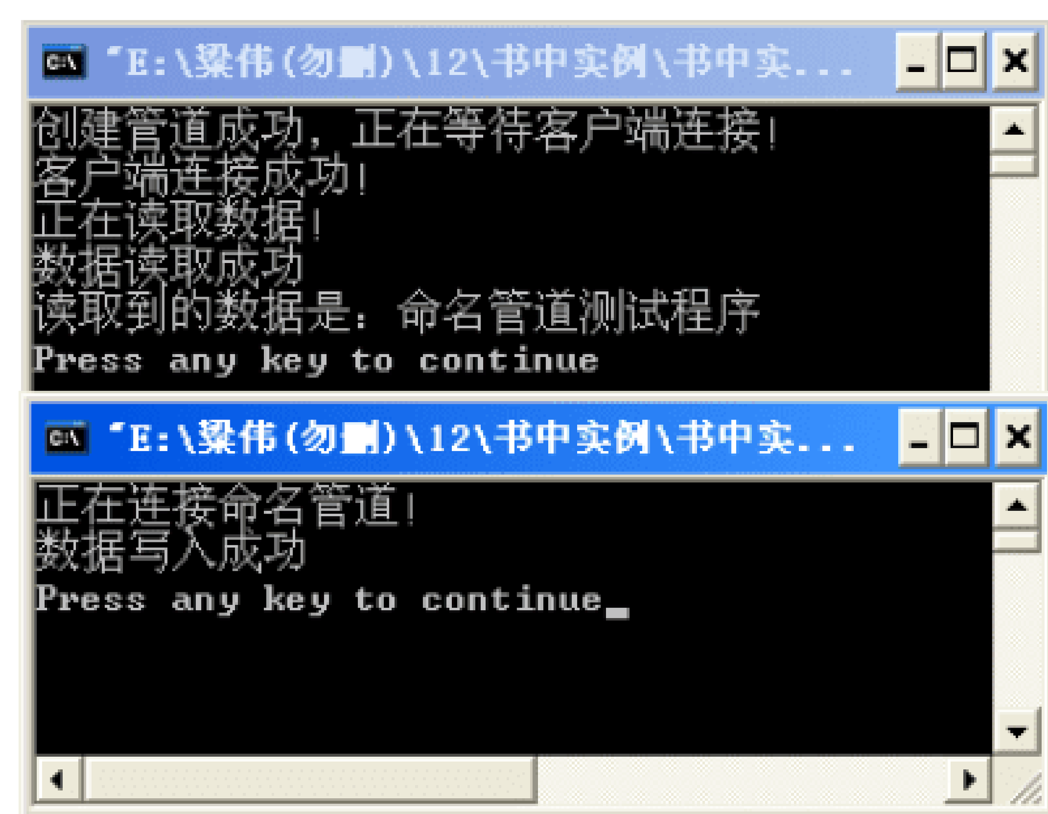


图 3.12　程序通过命名管道传输数据

3.3.3　匿名管道

匿名管道是没有命名的管道，只能被用在父进程与子进程之间进行数据通信。与命名管道相比，匿名管道不能被使用在网络进程之间。在本节中，将向用户讲解使用匿名管道进行数据传输的方法。

注意：子进程是指由父进程调用函数 CreateProcess()所创建的进程。

1．创建匿名管道

在程序中，用户可以调用函数 CreatePipe()创建匿名管道。该函数原型如下：

```
BOOL CreatePipe(
  PHANDLE hReadPipe,
  PHANDLE hWritePipe,
  LPSECURITY_ATTRIBUTES lpPipeAttributes,
  DWORD nSize
);
```

如果该函数调用成功，则返回 true，并将匿名管道的句柄放入用户指定的句柄变量中。否则，函数将返回 false。其参数及意义如下：

- ❑ 参数 hReadPipe 表示匿名管道的读取句柄。
- ❑ 参数 hWritePipe 表示匿名管道的写入句柄。

注意：*以上两个参数均是该函数需要返回的读写句柄。在子进程中，也就是使用这两个句柄通过匿名管道与父进程进行通信。*

- ❑ 参数 lpPipeAttributes 是指向结构体 SECURITY_ATTRIBUTES 的指针，表示匿名管道的安全属性。由于在匿名管道中，子进程需要继承父进程的读写句柄，所以不能设置该参数为 NULL。因此用户在实际编程时，应该初始化结构体 SECURITY_ATTRIBUTES 中的成员。该结构定义如下：

```
typedef struct _SECURITY_ATTRIBUTES {
    DWORD nLength;                    //指定该结构体的大小
    LPVOID lpSecurityDescriptor;      //安全描述符。一般情况下，用户将该成员设置
                                        为 NULL
    BOOL bInheritHandle;              //表示该进程所返回的句柄是否能被一个新进程所
                                        继承
} SECURITY_ATTRIBUTES;
```

在该结构体中，最重要的成员是第三个。其决定了匿名管道的读写句柄是否能被子进程所继承，所以用户必须将该成员设置为 true。

- ❑ 参数 nSize 表示匿名管道缓冲区的大小。若该参数为 0，则表示系统将使用默认的缓冲区大小。

例如，用户使用函数 CreatePipe()创建一个匿名管道。代码如下：

```
...                                 //省略部分代码
SECURITY_ATTRIBUTES ss;             //定义结构体 SECURITY_ATTRIBUTES 变量
ss. nLength=sizeof(ss);             //填充结构体中的各成员
ss. lpSecurityDescriptor=NULL;
ss. bInheritHandle=TRUE;
HANDLE read,write;                  //定义读写句柄
if(CreatePipe(&read,&write,&ss,0))  //创建匿名管道
{
   MessageBox("创建匿名管道成功");
}
```

在代码中，用户需要特别注意在填充结构体 SECURITY_ATTRIBUTES 的成员时，必须将成员 bInheritHandle 设置为 true。否则，子进程将无法继承父进程的读写句柄。

2. 创建子进程

当用户创建匿名管道成功后，便可以调用函数 CreateProcess()创建子进程。该函数原型如下：

```
BOOL CreateProcess(
  LPCTSTR lpApplicationName,
  LPTSTR lpCommandLine,
  LPSECURITY_ATTRIBUTES lpProcessAttributes,
  LPSECURITY_ATTRIBUTES lpThreadAttributes,
  BOOL bInheritHandles,
  DWORD dwCreationFlags,
  LPVOID lpEnvironment,
  LPCTSTR lpCurrentDirectory,
  LPSTARTUPINFO lpStartupInfo,
  LPPROCESS_INFORMATION lpProcessInformation
);
```

该函数如果调用成功，则返回 true。否则，函数将返回 false。其参数及意义如下：

- 参数 lpApplicationName 表示启动进程的完整路径。如果用户没有为该参数指定完整路径，则函数将在当前目录下搜索启动进程的可执行文件。

注意：当用户使用该参数时，必须加上可执行文件的扩展名。否则，函数将不会主动为其添加扩展名。

- 参数 lpCommandLine 表示传递给新进程的命令行参数。用户在编程时，可以在该参数中指定启动进程的路径。如果用户所指定的进程路径不是一个完整的路径，则函数将在系统的搜索路径下进行搜索并将自动为该启动进程的文件添加扩展名“.exe”。
- 参数 lpProcessAttributes 和 lpThreadAttributes 分别表示启动进程的进程对象以及该进程主线程的安全属性。若用户使用默认的安全属性，则将这两个参数分别设置为 NULL。
- 参数 bInheritHandles 表示启动进程是否能够继承父进程的相关句柄。当用户使用匿名管道编程时，必须将该参数设置为 true。
- 参数 dwCreationFlags 表示启动进程创建时的附加标记。由于在本章中用户仅仅是调用该函数启动一个进程，所以用户将该参数设置为 0 即可。
- 参数 lpEnvironment 表示启动进程所运行的内存环境。如果该参数为 NULL，则表示启动进程将使用调用进程（父进程）的内存环境。

注意：在实例程序中，用户将该参数直接设置为 NULL 即可。

- 参数 lpCurrentDirectory 指定启动进程运行后的路径，该路径必须是完整的路径名。如果该参数为 NULL，则表示子进程与父进程共用相同的路径。
- 参数 lpStartupInfo 是指向结构体 STARTUPINFO 的指针，表示启动进程将如何显示。该结构如下：

```
typedef struct _STARTUPINFO {
    DWORD   cb;                     //该结构体的大小
    ...                             //省略部分成员
```

```
    DWORD   dwFlags;              //指定该结构体中哪些成员可用
    HANDLE  hStdInput;            //指定读取句柄
    HANDLE  hStdOutput;           //指定写入句柄
    HANDLE  hStdError;            //指定错误句柄
} STARTUPINFO, *LPSTARTUPINFO;
```

由于该结构体的成员非常多，所以在这里只向用户讲解在匿名管道编程中需要使用的几个成员。其中，成员 dwFlags 决定了该结构体中哪些成员可用，如果将其指定为 STARTF_USESHOWWINDOW，则表示结构体中的成员 hStdInput、hStdOutput 和 hStdError 可用。用户也可以通过调用函数 GetStdHandle()获得系统中标准输入、输出和错误句柄。函数 GetStdHandle()的原型如下：

```
HANDLE GetStdHandle(
  DWORD nStdHandle
);
```

其中，参数 nStdHandle 表示用户需要获得的句柄类型，其值如表 3.6 所示。

表 3.6　获取的句柄类型

取　　值	意　　义
STD_INPUT_HANDLE	系统标准输入句柄
STD_OUTPUT_HANDLE	系统标准输出句柄
STD_ERROR_HANDLE	系统标准错误句柄

例如，用户在程序中填充结构体 STARTUPINFO 中的各个成员。代码如下：

```
...                                                  //省略部分代码
STARTUPINFO sa={0};                                  //定义并初始化结构体
sa.cb=sizeof(sa);                                    //填充结构体中的各个成员
sa.dwFlags=STARTF_USESHOWWINDOW;
sa. hStdInput=read;
sa. hStdOutput=write;
sa. hStdError= GetStdHandle(STD_ERROR_HANDLE);
```

- 参数 lpProcessInformation 是指向结构体 PROCESS_INFORMATION 的指针。该参数主要用于接收新进程的相关信息。

例如，用户在程序中使用函数 CreateProcess()创建一个可继承读写句柄的子进程。代码如下：

```
PROCESS_INFORMATION pp={0};                          //定义并初始化结构
...                                                  //省略部分代码
CreateProcess(NULL,"子进程.exe",NULL,NULL,TRUE,0,NULL,NULL,&sa,&pp);
                                                     //创建子进程
```

首先，用户定义并初始化结构体 PROCESS_INFORMATION 变量。然后调用函数 CreateProcess()创建子进程，并将子进程的相关信息保存在变量 pp 中。

注意：用户创建子进程成功以后，可以调用函数 ReadFile()和 WriteFile()对匿名管道进行数据读取和数据写入。

3. 父进程实例

在本小节中，将通过实现父进程实例程序向用户讲解匿名管道通信中父进程端的具体

实现方法。首先，在 VC 中创建基于控制台窗口的工程，名称修改为“匿名管道”。然后在该工程中，添加一个 C++源文件，名称修改为“父进程”。然后在源文件中添加代码实现其功能，代码如下：

```
#include<windows.h>
#include<stdio.h>
main()
{
 HANDLE read=NULL,write=NULL;                //定义读写句柄
 SECURITY_ATTRIBUTES ss;                     //定义结构体 SECURITY_ATTRIBUTES 变量
 STARTUPINFO sa={0};                         //定义结构体 STARTUPINFO 变量
 PROCESS_INFORMATION pp={0};                 //定义结构体 PROCESS_INFORMATION 变量
 char text[]="匿名管道程序测试！";             //定义并初始化字符数组
 DWORD writetext;
ss.nLength=sizeof(ss);                       //填充结构体中的各成员
ss.lpSecurityDescriptor=NULL;
ss.bInheritHandle=TRUE;
if(CreatePipe(&read,&write,&ss,0))           //创建匿名管道
{
   printf("创建匿名管道成功\r\n");             //定义并初始化结构体
sa.cb=sizeof(sa);                            //填充结构体中的各个成员
sa.dwFlags=STARTF_USESHOWWINDOW;
sa. hStdInput=read;
sa. hStdOutput=write;
sa. hStdError= GetStdHandle(STD_ERROR_HANDLE);
if(::CreateProcess(NULL," 子 进 程 .exe",NULL,NULL,TRUE,0,NULL,NULL,&sa,
&pp))
                                             //创建子进程
{
 printf("创建子进程成功\r\n");
 WriteFile(write,text,sizeof(text),&writetext,NULL);
                                             //将数据写入匿名管道
printf("通过匿名管道写入数据成功\r\n");
}
Else                                         //写入失败
{
printf("创建子进程失败\r\n");
        }
 }
}
```

将上面的代码编译，生成父进程实例。

4．子进程实例

现在，用户需要编程实现子进程程序实例。首先，在工程“匿名管道”中添加一个 C++源文件，名称修改为“子进程”。然后在该源文件中添加代码实现其功能，代码如下：

```
#include<windows.h>                              //包含头文件
#include<stdio.h>
main()                                           //主函数
{
 HANDLE read=NULL;                               //定义读取句柄
 char text[100]={0};                             //定义并初始化字符数组
 DWORD readtext;
 read=GetStdHandle(STD_INPUT_HANDLE);            //获取读取句柄
```

```
if(ReadFile(read,text,100,&readtext,NULL))          //读取匿名管道中的数据
{
 printf("从匿名管道中读取的数据是：%s\r\n",text);      //输出数据
}
Sleep(10000);                                        //程序睡眠 10 秒钟
return 0;                                            //程序正常返回
}
```

将程序编译，执行子进程实例。然后，用户可以首先打开父进程，程序打开以后会自动创建匿名管道和子进程。通过匿名管道通信的效果，如图 3.13 所示。

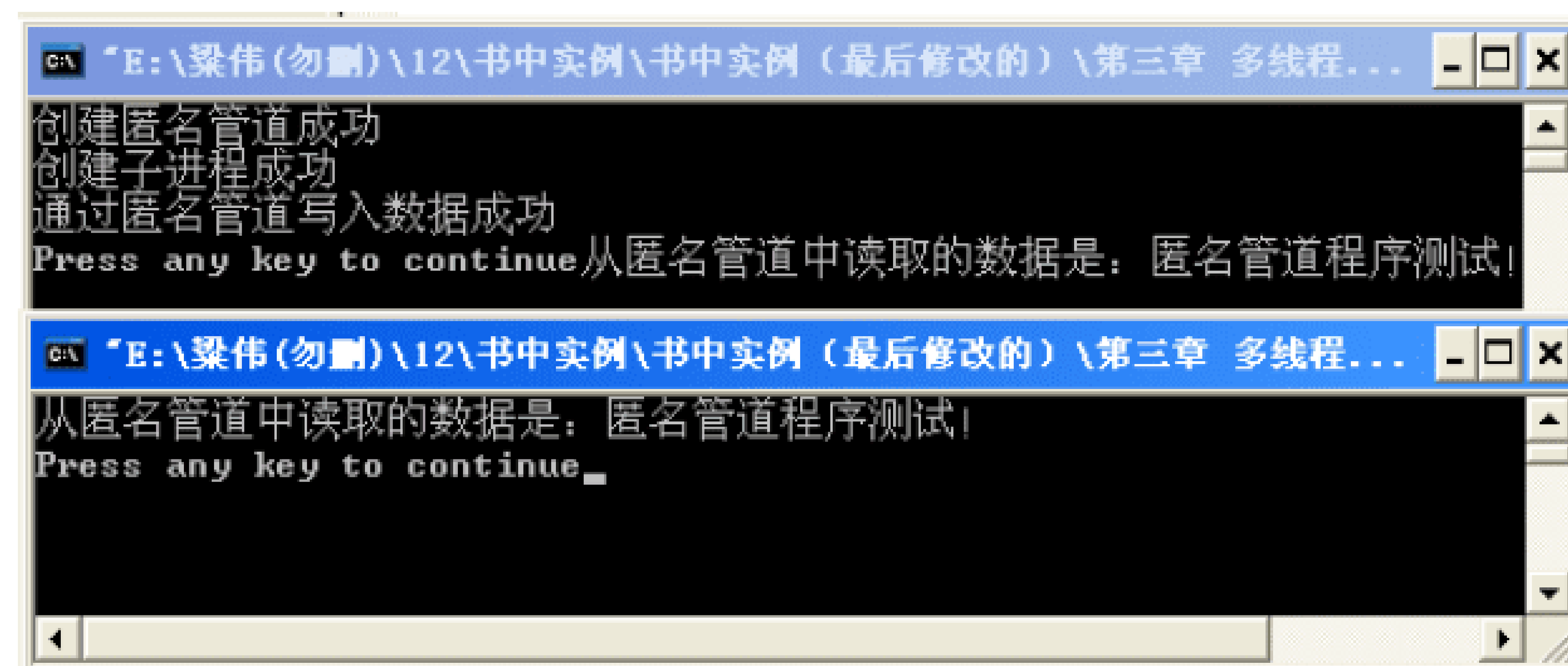


图 3.13　使用匿名管道实现进程间通信

3.3.4　小结

本节主要向用户讲解了实现进程间通信的几种常见方法。其中，邮槽以及命名管道不但可以使用在本地进程之间，还可以使用在网络进程之间的数据通信。而匿名管道只能使用在本地进程之间的数据通信。通过本节的学习，用户对进程间通信的方式以及实现方法可以有一个进一步的了解。

3.4　设置 I/O 模式

在本章知识中，主要向用户讲解网络套接字的异步 I/O 模式。通常情况下，当用户使用网络套接字进行程序编写时，为了提高程序的运行效率，则应该使程序在有相关的事件响应时才实现其功能。否则，没有相关事件发生时，则应用程序处于后台运行状态。这样，可以使用户计算机的运行效率大大提高。本节将着重向用户讲解如何设置套接字的 I/O 模式。

3.4.1　异步 I/O 模式

在套接字编程中，异步 I/O 模式是指当网络中有相关的套接字消息到来时，程序才会调用相关的响应函数对该消息进行处理。否则，程序将在系统后台继续等待相关的消息到来或者实现其他操作。

例如，一个异步套接字程序处理的套接字消息是连接和接收。那么，当该程序在所创

建的套接字上监测到有连接消息时，程序会调用连接消息的响应函数对该消息进行相关处理；如果监测到的套接字消息是接收消息，其处理过程也一样。当套接字处理完已经监测到的消息以后，程序会在系统后台中继续监测套接字相关消息。这样，不仅可以降低程序对系统资源的使用频率，还能提高程序的执行效率。

如果用户使用 VC 编写异步套接字程序，可以调用函数 WSAAsyncSelect()将套接字设置为异步模式。关于该函数的具体讲解将在 3.4.2 节中进行。

3.4.2　WSAAsyncSelect 方法

函数 WSAAsyncSelect()的作用是将用户指定的套接字对象设置为异步模式。该函数的原型如下：

```
int WSAAsyncSelect (
  SOCKET s,
  HWND hWnd,
  unsigned int wMsg,
  long lEvent
);
```

参数如下：

- 参数 s 表示需要设置为异步模式的套接字句柄。
- 参数 hWnd 表示接收消息响应的窗口句柄。
- 参数 wMsg 表示响应消息标识。
- 参数 lEvent 表示发生在该套接字上的事件，其取值如表 3.7 所示。

表 3.7　套接字事件部分标识及其意义

取　值	意　义
FD_READ	套接字上发生读取事件
FD_WRITE	套接字上发生写入事件
FD_ACCEPT	套接字上发生连接事件
FD_CLOSE	套接字上发生关闭事件

在使用该函数设置异步套接字之前，用户首先需要定义一个自定义消息并为其关联消息响应函数。例如，在本节中将定义消息 WM_SOCKET，与该消息关联的消息响应函数为 OnSocket()。然后，用户在程序初始化函数中使用函数 WSAAsyncSelect()将套接字设置为异步模式。代码如下：

```
...                                                    //省略部分代码
WSAAsyncSelect(s,this->m_hWnd, WM_SOCKET, FD_ACCEPT| FD_READ)
                                                       //设置异步套接字
```

在上面的代码中，用户设置异步套接字的同时指定了套接字消息需要处理的相关事件。如果用户将异步套接字设置成功后，需要实现套接字消息响应函数 OnSocket()。代码如下：

```
void CMy2Dlg::Onsockt1(WPARAM wParam,LPARAM lParam)
 {
  switch(lParam)
```

```
{
 case FD READ:                                      //处理套接字接收事件
    ...                                             //省略部分代码
 case FD ACCEPT:                                    //处理套接字连接事件
    ...                                             //省略部分代码
}
 }
```

在上面的代码中，用户根据消息参数 lParam 判断具体发生的套接字事件，然后再根据该事件进行相应的处理。由于在本节中，主要向用户讲解函数 WSAAsyncSelect()的相关用法，所以关于套接字消息响应函数的具体实现将在后面的章节中进行具体讲解。

3.5 小　　结

在本章中，主要介绍了多线程程序的工作原理以及多线程程序的设计方法。并且通过线程同步技术向用户讲解互斥对象等线程同步方法的具体实现。在进程通信中，列举了几种比较常见的通信方式。例如，邮槽、命名管道等。同时，邮槽与命名管道不但能在本地进程之间进行数据通信，还能在网络进程之间进行数据通信。在 3.4 节中，还向用户讲解了异步套接字的相关函数以及实现方法。

通过本章相关内容的学习，用户应该掌握在 Windows 编程中怎样实现多线程程序、线程同步、进程间通信等程序设计方法。本章中的具体代码请参考随书光盘中的相关内容。

第 2 篇　Visual C++网络编程典型应用

第 4 章　FTP 浏览器

FTP 是一种文本传输控制协议。其常常被用来在不同的计算机之间传输文本文件。用户通过 FTP 浏览器向服务器发送各种命令，可以很方便地查看远程计算机上的文件等信息。用户既可以把文件从远程计算机复制到本地计算机，还可以从本地计算机复制到远程计算机。本章将详细讲解以上功能的 VC 实现。

4.1　FTP 工作原理

FTP 的工作原理跟 TCP 一样，客户端需要先与服务器连接，等待服务器的应答，最后再建立数据通道。所以，FTP 浏览器在和服务器建立连接时也需要经过“三次握手”的过程。这表示客户端与服务器之间的连接是可靠、安全的，这也为数据传输提供了可靠的保证。FTP 的工作原理如图 4.1 所示。

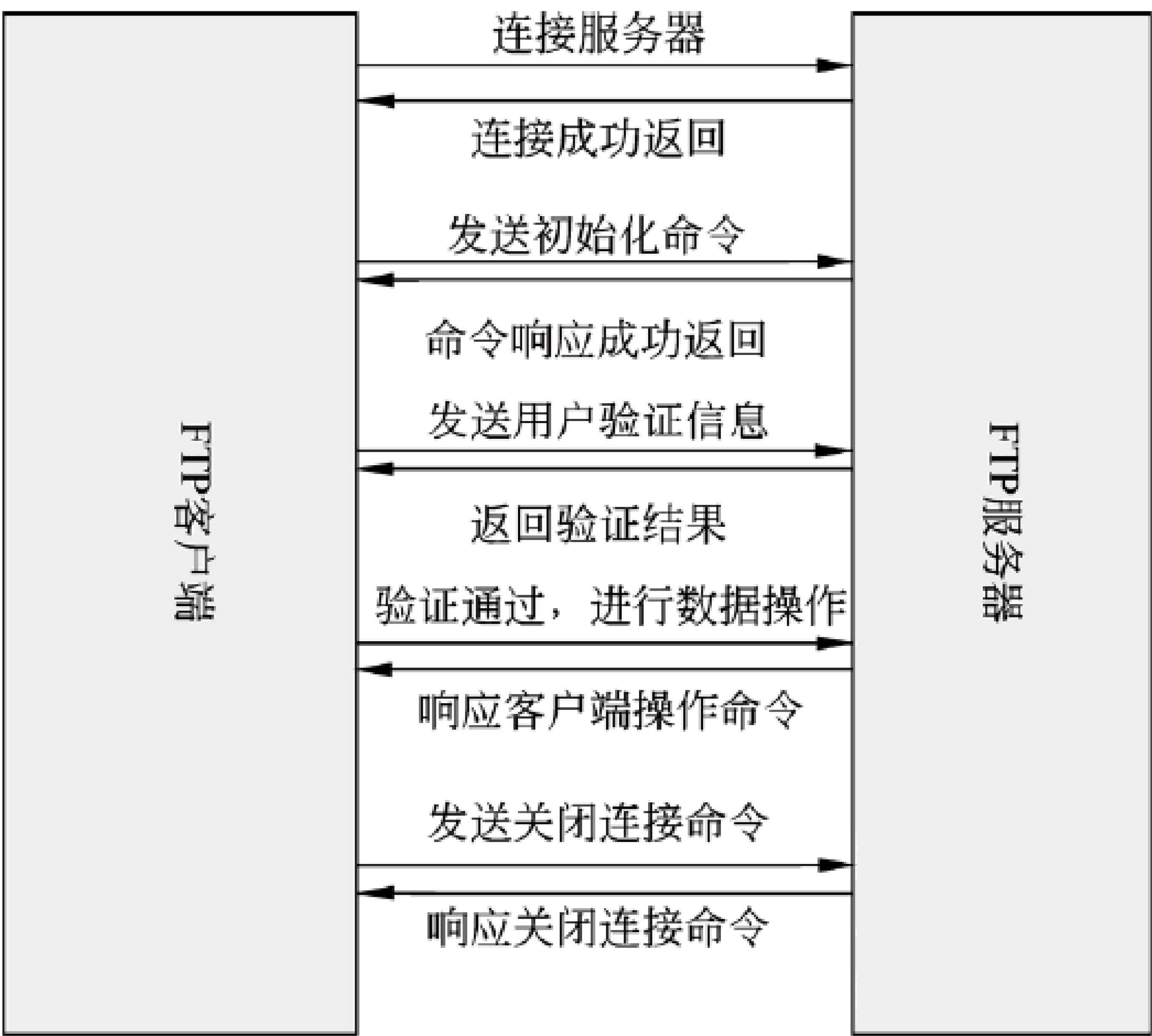


图 4.1　FTP 工作原理图

4.1.1　FTP 数据结构

进行 FTP 编程之前，用户首先需要知道 FTP 有哪些数据结构。由于在某些主机上保存的文件是面向字节的，某些是面向记录的。所以在 FTP 中，除了有不同的数据类型以外，还有几种不同的文件结构类型。这样做的目的是为了在不同的主机之间传送文件时能够相

互识别。

- ❑ 二进制结构：文件中没有内部结构，一般被看作二进制流。
- ❑ 文件式结构：由许多记录组成的文件。
- ❑ 页面结构：由不同的索引页组成文件。

注意：一般情况下，如果没有使用 FTP 命令去设置文件的结构，则默认的结构是文件式结构。

4.1.2　FTP 数据传输模式

在 FTP 的数据传输中，传输模式将决定文件数据会以什么方式被发送出去。一般情况下，网络传输模式有 3 种：将数据格式化后传送、压缩后传送、不做任何处理进行传送。当然不论用什么模式进行传送，在数据的结尾处都是以 EOF 结束。在 FTP 中定义的传输模式有以下几种。

1．二进制模式

二进制模式就是将发送数据的内容转换为二进制表示后再进行传送。这种传输模式下没有数据结构类型的限制。

在二进制结构中，发送方发送完数据后，会在关闭连接时标记 EOF。如果是文件结构，EOF 被表示为双字节。其中第一个字节为 0，而控制信息包含在后一个字节内。

本书中如无特别说明，均采用该模式进行传输数据。

2．文件模式

文件模式就是以文件结构的形式进行数据传输。文件结构是指用一些特定标记来描述文件的属性以及内容。一般情况下，文件结构都有自己的信息头，其中包括计数信息和描述信息。信息头大多以结构体的形式出现。

- ❑ 计数信息：计数指明了文件结构中的字节总数。
- ❑ 描述信息：描述信息是负责对文件结构中的一些数据进行描述。例如，其中的数据校验标记是为了在不同主机间交换特定的数据时，不论本地文件是否发生错误都进行发送。但在发送时发送方需要给出校验码，以确定数据发送到接收方时的完整性、准确性。

在文件结构中，既可以用记录结构，也可以用相对应的数据表示。文件的信息头结构如表 4.1 所示。

表 4.1　文件的信息头结构

文件信息头计数信息大小	文件信息头描述信息大小
计数信息占 16 位字节	描述信息占 8 位字节

描述信息是由字节中的位特定标记值来说明。列举几个特定标记值及其意义，如表 4.2 所示。

表 4.2　特定标记值及意义

标　记　值	意　　义
64	表示文件的结束符标记 EOF
32	表示文件中有可疑错误
16	表示具有重发标记的文件

由表 4.2 可知，描述信息中可能存在多个标记值，所以必须将需要用到的标记都进行设置。

3．压缩模式

在这种模式下，需要传送的信息包括一般数据、压缩数据和控制命令。

- 一般数据：以字节的形式进行传送。
- 压缩数据：包括数据副本和数据过滤器。
- 控制命令：用两个转义字符进行传送。

注意：此种传输模式请参考其他相关书籍，本书不再进行深入讲解。

在 FTP 数据传输时，发送方必须把数据转换为文件结构指定的形式再传送出去，而接收方则相反。因为进行这样的转换很慢，所以一般在相同的系统中传送文本文件时都采用二进制流表示比较合适。

4.1.3　与服务器进行连接

FTP 客户端需要与服务器连接成功后，才能进行文件数据的传输。当连接时，客户端需要用户指定端口、连接模式等操作。

1．连接所使用的端口

在连接端口的使用上，FTP 与 HTTP 不同。因为 FTP 在与服务器连接时需要用到两个端口：其中一个端口（FTP 的默认端口是 21）作为控制连接端口，它主要用于发送命令给服务器以及等待服务器的响应；另一个端口是数据传输端口，端口号为 20 或者任意有效端口号，用来建立数据传送通道。

2．连接模式

FTP 客户端连接服务器的模式有两种：PORT 模式和 PASV 模式。

- PORT 模式：PORT 是主动模式。当客户端选择这种模式与服务器进行连接的时候，它需要向服务器提供一个 IP 地址和一个端口号。
- PASV 模式：PASV 是被动模式。当选择这种模式连接时，服务器需要提供给客户端一个 IP 地址和一个端口号。用户平时从网上一个指定的 FTP 地址和端口下载文件就是这种模式的一种实际应用，相反则为 PORT 模式。

注意：在本章中如无特别说明，所选用的连接模式均是主动模式。

4.1.4　登录验证

在连接 FTP 服务器成功之后，用户需要发送相关命令或者是数据流到服务器进行身份验证或其他操作。在本章的 4.1.6 小节中，给出了一些常用的 FTP 命令。

1．登录方式

在登录 FTP 时，登录方式有匿名登录、代理登录或者是通过用户名登录等。各种登录方式的不同在于访问文件的权限（只读、只写或者读写），这也是 FTP 的一个重要特点。

注意：*在本章中涉及到的登录方式主要是以用户名登录为主。*

2．验证

客户端将用户名和密码以命令的方式发送到服务器进行验证，例如，用户名为“lymlrl”，密码为“123456”的用户在进行验证时，将其转换成命令流：“USER”+lymlrl+“PASS”+123456；这个命令将作为字符串被发送到服务器，这个工作是通过 CArchive 等类中的函数实现的（具体内容将在 4.3 节中讲解）。

服务器在验证之后会返回结果给客户端。如果返回值的第一个数字为 1、2 或者是 3，则表示返回值正确，否则发生错误。然后提取当前位置的下一条命令值，如果为 EROR 表示出现用户名或密码错误；为 SUSS 则表示验证成功。

4.1.5　关闭数据连接

通常情况下，服务器只负责进行数据连接，并对它进行初始化和关闭。除非客户端在命令控制中主动要求关闭连接时，服务器才会关闭连接。当然服务器也会在以下情况下关闭数据连接。

- 当服务器发送数据结束时，会通过 EOF 终止传送；
- 客户端发送 ABORT 命令；
- 客户端改变了端口号；
- 控制连接通道被关闭；
- 传输过程中发生严重错误。

但是，在一般情况下客户端与服务器之间的连接都是在数据正常处理完成以后关闭的。

4.1.6　FTP 常用命令

在实际编程中，有些复杂的操作，只是需要客户端发送相关的指令到服务器执行即可。所以，对于用户来说掌握常用的 FTP 命令是非常重要的。下面列举了一些常用的 FTP 命令，如表 4.3 所示。

表 4.3　常用FTP命令及意义

FTP 命令	意　　义
LIST	发送当前工作目录下的文件名列表到客户端
PWD	显示服务器的当前工作目录名
RETR	从服务器下载一个文件
STOR	上传文本文件到服务器，如果文件存在会被覆盖
*STOU	上传文本文件到服务器，但不会覆盖已经存在的文件
STRU	设置文件的结构
MODE	指定数据的传输模式
ABORT	通知服务器关闭连接

在表 4.3 中，已经列举了部分常用的 FTP 命令。通常情况下，客户端通过 CArchive 类的成员函数 WriteString()可以将这些命令以字符串的形式发送到服务器执行。然后，客户端使用 CArchive 类的成员函数 ReadString()来获取服务器返回的数据。关于这两个函数的一些用法将在下一节实例中进行讲解。

4.1.7　数据校验与重发控制

FTP 是属于 TCP/IP 簇中的一种具体应用，所以 FTP 也具有数据重发机制。但在 FTP 中，数据重发仅用于文件和压缩模式。一般情况下，重发机制都要求发送者在发送数据时加入特定标记来描述数据的重要信息。并且该标记只针对发送者有意义，其内容大多是用来校验数据的完整性。特定标记可以表示任何可以标记的属性或其他信息。

如果接收方也支持重发机制，那么接收方系统中将会保存这一特定标记。当系统重新启动或者其他原因造成系统重启，用户均可以根据原来的标记继续传送数据。其实，用户经常用到的断点续传就是很好的一个例子。当接收方收到一段数据后，记下标记，如果传送过程中出现错误，那么发送方将会从这个标记点重新传送数据。

4.2　登录 FTP 服务器

在对 FTP 文件进行相关处理之前，用户必须在成功连接、登录服务器以后，才可以执行相关的操作。本节将主要讲述 FTP 的连接以及登录验证过程，选择的连接模式是 PASV（被动）模式，登录方式是用户名登录。

4.2.1　连接 FTP 服务器

因为 FTP 连接是基于 Windows 套接字编程的，所以 FTP 的连接过程和 Socket 连接一样。也就是客户端创建连接套接字以后，调用函数 Connect()向服务器发送连接请求。用户需要特别注意的是待服务器同意连接并返回后，客户端必须先发送一个空字符串到服务器进行初始化，这样才能进行数据的交换。在连接 FTP 服务器程序的开发中，用户需要用到 MFC 的一些类或函数，如 CSocket 类。

1．实例化CSocket

CSocket 类是 MFC 对 Windows 套接字的一个封装类。一般，它使用类中的构造函数来实例化对象。但是在实际编程中，构造函数常使用 new 关键字来调用，构造一个没有初始化的套接字对象。例如，代码如下：

```
CSocket m_clientsocket=new CSocket;    //构造一个套接字对象m_clientsocket
```

构造该对象后，必须调用成员函数 Create()创建完整的套接字句柄。其函数原型为：

```
BOOL CSocket::Create(UINT nSocketPort,
int nSocketType=SOCK_STREAM, LPCTSTR lpszSocketAddress=NULL);
```

在该函数中，参数 nSockPort 用来指定与套接字相关联的本地端口号，默认值为 0，本章实例中均设置为 FTP 的默认端口 21。nSocketType 指定套接字类型，默认为 SOCK_STREAM 表示创建流式套接字，FTP 客户端均采用默认类型。lpszSocketAddress 指定创建套接字的网络地址，默认为 NULL，客户端一般设置为 NULL。该函数调用成功返回非 0 值，否则返回 0，表示出错。

2．实现连接功能

用户创建套接字对象以后，就可以调用函数去实现真正的连接了。在 CSocket 类中，实现连接功能的函数是 Connect()，其函数原型如下：

```
BOOL Connect(LPCSTR lpszHostAddress, UINT nHostPort);
```

参数 lpszHostAddress 指定将要连接的服务器地址；nHostPort 指定将要连接服务器上的端口号。注意，一般在 FTP 编程中都将此参数设置为 21 号端口。该函数连接服务器成功，则返回 true，否则返回 false。例如，用户需要连接一个 IP 地址为“218.6.132.5”的服务器，则其代码如下：

```
CSocket  *m_clientsocket=new CSocket();            //构造连接套接字对象
m_clientsocket->Create(21, SOCK_STREAM, FD_READ|FD_WRITE, NULL);
                                                   //创建流式套接字
m_clientsocket->Connect("218.6.132.5", 21);        //连接服务器
```

这段代码的作用是创建套接字对象以后，调用函数 Connect()进行连接服务器的操作。

3．封装连接过程

本节中，FTP 的连接过程主要由自定义函数 FTPConnect()实现。函数 FTPConnect()的作用是根据用户输入的服务器地址和端口号，连接 FTP 服务器。其返回值表示操作是否成功。其具体代码如下：

```
//连接服务器函数，参数 severhost 表示服务器 IP 地址，port 指定要连接的服务器端口
BOOL FTPConnect(CString severhost, int port)
{
CSocket  *m_clientsocket=new CSocket();        //构造连接套接字对象
m_clientsocket->Create(21, SOCK_STREAM, FD_READ|FD_WRITE, NULL);
if(!m_clientsocket)                            //判断套接字对象创建是否成功
```

```
{ MessageBox("套接字创建失败！");
return false; }                              //创建 m_clientsocket 失败
  if(!(m_clientsocket->Connect(severhost, port))) return false;
     //连接 FTP 服务器
    else
{ return true;}                              //连接成功将返回 true
}
```

在该函数中，参数 severhost 和 port 分别指定要连接服务器的地址和端口。程序首先创建流式套接字 m_clientsocket，然后调用 CSocket 类的函数 Connect()连接 FTP 服务器。当连接失败时，函数返回 false；否则，返回 true。

自定义函数 FTPConnect()封装完成后便可以对其进行调用。代码如下：

```
Void OnConnect()
{ CString address="218.6.132.5";             //定义 IP 地址字符串变量并初始化
  int port=21;                               //定义端口变量并初始化
if(!(FTPConnect((LPCSTR)address,port)))      //判断 FTPConnect 调用是否成功
{  MessageBox("连接失败！");}                 //提示失败
else
{MessageBox("连接服务器成功！");}              //否则提示成功
}
```

函数 OnConnect()根据定义的 IP 地址和端口号，调用自定义函数 FTPConnect()连接服务器。其中，IP 地址和端口号可以由用户进行自定义。以上代码实现了客户端连接服务器的基本功能。

4.2.2　登录 FTP 服务器

对于客户端而言，连接服务器成功后需要发送用户名等验证信息到服务器进行验证登录。

1．构造验证信息

在 FTP 中，验证信息是以命令字符串的形式发送到服务器的。首先，验证信息由用户名和密码组成。用户名用命令 USER 标识，密码用命令 PASS 标识。例如，用户名为“lymlrl”，密码为“123456”的验证信息写成命令流为“USER”+lymlrl+“PASS”+123456。

如果用户使用匿名登录方式登录 FTP 服务器，则用户名使用默认的 anonymous，密码可以是自己的邮箱。那么命令流为“USER”+anonymous+“PASS”+xxxx@163.com。

2．发送验证信息

验证信息构造完成后，用户可以使用 MFC 类库中的 CSocketFile 和 CAchive 两个类进行数据发送。一般情况下，CSocketFile 类是和 CSocket 类一起使用的，它主要用来创建一个与套接字相关联的文件对象。例如，创建一个与 4.2.1 节中套接字对象 m_clientsocket 相关联的 CSocketFile 对象，代码如下：

```
CSocketFile *m_sockfile;                     //定义文件对象指针
m_sockfile=new CSocketFile(m_clientsocket);
                              //关联对象，m_clientsocket 是创建的套接字
```

然后，再使用 CAchive 类创建一个与该文件指针对象 m_sockfile 关联的串行化对象。代码如下：

```
CArchive *m_archive;                           //定义串行化指针对象
m_archive=new CArchive(&m_sockfile, CArchive::load| CArchive::store);
                                               //创建对象 m_archive 的实例并指定属性
```

一般情况下，用户在创建串行化对象 m_archive 时，都要为其指定属性 CArchive::load| CArchive::store，表示创建的对象 m_archive 具有读取和存储的功能。

现在，用户可以调用 CArchive 类的 WriteString()函数进行命令发送。其原型如下：

```
Void WriteString(LPCTSTR lpsz);
```

参数 lpsz 指定将要操作的字符串。该函数调用失败会抛出一个异常。例如，用户要获取服务器上的文件目录需要向服务器发送 LIST 命令，使用该函数进行发送。代码如下：

```
m_archive->WriteString("LIST"+"\r\n");
                               //调用 CArchive 类的 WriteString 发送 LIST 命令
```

🔔**注意**：*有关 FTP 的其他命令请参考表 4.3。*

使用用户名和密码登录 FTP 服务器。其代码如下：

```
CString str=""USER"+lymlrl+"PASS"+123456";      //定义命令字符串变量
try
{
m_archive->WriteString(str+"\r\n");//调用 CArchive 类的 WriteString 发送命令
m_archive->Flush();                             //强制写入命令到服务器
}
  Catch(CException e)                           //处理被抛出的异常
 {
   MessageBox("发送关闭命令失败！");
  }
...                                             //省略部分代码
```

在代码中，所使用的登录方式是用户名密码登录。如果用户选择匿名登录，则将字符串变量初始化为“USER”+anonymous+“PASS”+xxxx@163.com 即可。然后使用函数 WriteString()发送到服务器，如果失败该函数则抛出异常进行处理。

用户在实际编程中，需要首先发送信息到服务器进行初始化，才能发送其他命令到服务器执行。具体代码如下：

```
CSocketFile *m_sockfile;                       //定义文件对象指针
CArchive *m_archive;                           //定义串行化指针对象 m_archive
m_sockfile=new CSocketFile(m_clientsocket);    //关联对象
m_archive=new CArchive(&m_sockfile, CArchive::load| CArchive::store);
                                               //创建对象 m_archive 的实例并指定属性
Void Send()                                    // Send()函数发送信息到服务器
{     CString charstring;
      charstring=""USER"+lymlrl+"PASS"+123456"; //构造字符串 charstring
      m_archive->WriteString(" "+"\r\n");    //向服务器发送空字符串进行初始化
 try{
```

```
m_archive->WriteString(charstring +"\r\n");
                                    //调用 CArchive 类的 WriteString 发送命令
m_archive->Flush();                 //强制写入命令到服务器
}
  Catch(CException e)               //处理被抛出的异常
 {
MessageBox("发送关闭命令失败！");
   }
}
```

在上面的示例代码中，程序首先调用函数 WriteString()发送一个空字符串到 FTP 服务器进行初始化工作。然后发送已经构造好的用户登录信息到服务器进行验证。

注意：在发送字符串的时候一定要在后面加上“\r\n”。

3．接收验证信息

客户端接收服务器返回的内容是通过 CArchive 类的函数 ReadString()进行的。这个函数的作用是从与 CArchive 对象相关联的套接字文件中读取数据到指定变量中。其原型如下：

```
BOOL ReadString( CString& rString );
```

其中，参数 rString 是存放接收数据的。该函数调用成功返回 true，否则返回 false。在本章中封装一个自定义函数 Recv()来接收服务器返回的数据。其功能代码如下：

```
CString Recv()                                  // Recv()函数接收服务器返回的数据
{
CString recvstr=" ";                            //初始化字符串 recvstr 为空
if(m_archive->ReadString(recvstr))              //接收返回信息并放到 recvstr 变量
{
if(recvstr==" ") MessageBox("接收数据为空");  //如果接收的数据为空则提示
{ MessageBox("接收数据成功");
return recvstr; }}                              //返回接收到的数据
else
 {
MessageBox("接收数据失败");                     //提示接收数据失败
   }
}
```

函数 Recv()的作用是接收服务器返回的数据，并将数据保存在字符串变量 recvstr 中，函数的返回类型是 CString 类型。在程序中，首先判断接收数据是否成功，再对接收到的数据进行判断是否为空。如果不为空，则直接返回接收到的数据。

用户接收到的数据内容，会因为 FTP 服务器的不同而不同。例如，有的服务器在客户端登录成功以后返回欢迎信息到客户端等。

4．分析数据

客户端在接收到服务器返回的信息以后，用户需要将返回的信息内容进行分析，以得到用户所需要的数据。该信息是一个字符串形式。它由 3 个数字、一个空格和一段文字信息组成。其形式如表 4.4 所示。

表 4.4　数据基本格式

3 个数字	空格	文字信息字符串

第一个数字若为 1 或者 2，则表示返回值正确。若为 3，则表示一个中间层的肯定回答，此时，服务器会等待客户端进一步的信息。若为其他，则表示错误。第二个数字表示回答的类型，若为 0，则表示语法错误。若为 1 则表示信息内容。第三个数字表示对错误进行具体的分类。

在文字信息中，如果内容为 EROR 表示出现用户名或密码错误。否则，表示验证成功。下面将使用自定义函数 Recv()来接收数据，然后再对接收到的数据进行分析。代码如下：

```
CString str1=" ";                  //初始化字符串变量，用于存放服务器返回的信息
char ch;                           //定义字符变量，用于字符比较
int i=0;                           //循环变量初始化为 0
str1=Recv();                       //调用 Recv()函数得到服务器返回的信息
if(str1.GetLength()>0)             //比较 str1 的长度
{
ch= str1.GetAt(i);                 //将接收到的第一个字符赋给 ch
}
if(ch==1&& ch==2)                  //如果返回的第一个字符等于 1 或者 2，表示返回值正确
{
 while(i<= str1.GetLength()&&ch!=NULL)
{  switch(ch)
   case EROR:                      //出现错误
        MessageBox("用户名或密码错误！");
        break;
   case SUSS:                      //验证成功
        MessageBox("登录成功！");
        return true;
break;
ch= str1.GetAt(++i);               //循环查看返回的信息
}
}
```

在代码中，用户通过 CString::GetAt(0)函数可以获得服务器返回信息的第一个字符。如果第一个字符等于 1 或 2，则表示返回值正确。否则，表示发生错误。最后，使用变量 i 进行循环获得命令值，如果是 EROR 表示验证错误。否则，表示成功。

5. 关闭与服务器的连接

当用户操作完需要的数据以后，需要关闭与服务器的连接。这时，向服务器发送 ABORT 或者是改变端口等操作都会使服务器关闭连接。在这里最为简便的方法是使用 CArchive 类的 WriteString()发送 ABORT 命令到服务器。关闭连接操作的关键代码如下：

```
try{                                   //尝试发送命令 ABORT 到服务器，以关闭连接
m archive->WriteString("ABORT "+"\r\n");
                                       //调用 CArchive 类的 WriteString 发送命令
m_archive->Flush();                    //强制写入命令到服务器
}
  Catch(CException e)
 {
MessageBox("发送关闭命令失败！");
    }                                  //抛出错误并处理错误
```

在这一小节中，用户主要了解了怎样去连接、登录 FTP 服务器、验证用户信息和关闭连接等过程实现。在上述操作都成功以后，用户就可以进行相关的文件处理操作了。关于 FTP 具体的文件处理将在下节中进行讲述。

4.3　FTP 文件处理

在 FTP 客户端编程中，用户可以通过一些类或者函数，获得服务器的文件目录和实现上传、下载文件等功能。例如，在 MFC 中，CSocketFile 类和 CArchive 类是在 FTP 编程中很重要的两个类。本节将重点介绍这两个类的一些用法和示例。

4.3.1　CSocketFile 类的使用

在 MFC 中定义了一个在套接字编程中使用的类：CSocketFile 类；它可以充分发挥 CSocket 类的特性。CSocketFile 类是 CFile 的派生类，它主要用来在 Windows Sockets 编程中发送和接收序列化数据（如结构体数据）。通过把 CSocketFile 类对象、CArchive 对象和 CSocket 创建的套接字对象联系起来，可以实现数据的加载（接收）和存储（发送）。

1．构造函数

在实际编程中，将 CSocketFile 对象和 CSocket 对象直接联系起来可以用 CSocketFile 类的构造函数来完成。CSocketFile 类构造函数原型如下：

```
CSocketFile::CSocketFile(CSocket* pSocket, BOOL bArchiveCompatible =
TRUE );
```

参数 pSocket 是一个 CSocket 对象；bArchiveCompatible 指示该文件对象是否与一个 CArchive 对象一起使用，默认为 true。该构造函数可以有两种调用方式。例如：

```
CSocket  *m_clientsocket=new CSocket;                //创建套接字
CSocketFile *m_sockfile=new CSocketFile(&m_clientsocket);
                                //创建一个与 m_clientsocket 关联的文件指针对象
```

上述代码中，是通过 new 关键字调用 CSocketFile 类的构造函数创建一个指针对象的。第二种调用方式如下：

```
CSocket  *m_clientsocket=new CSocket;           //创建套接字
CSocketFile  m sockfile(&m clientsocket);
                                    //创建一个与 m_clientsocket 关联的文件对象
```

这两种调用方式都需要在实例化对象 m_sockfile 之后，再与 CArchive 对象相关联，并由 CArchive 对象指定其属性。属性取值如表 4.3 所示。代码如下：

```
CArchive ar(&m_sockfile, CArchive::load| CArchive::store);
                                //创建与 m_sockfile 相关联的串行化对象并指定属性
...                             //省略部分代码
ar.Close();                     //关闭串行化对象
```

🔔注意：在这里使用完串行化对象 ar 以后，需要使用函数 CArchive::Close()关闭，确保数据（命令）及时存储（发送）。

2．CSocketFile 与CFile进行比较

CSocketFile 类虽然派生于 CFile 类，但是它却屏蔽掉了函数 CFile::Open()。也就是说，用户在实际编程时，不能使用 CSocketFile 对象直接去调用函数 Open()打开文件。

🔔注意：由于在本章实例中，有关文件的操作大多是由 CArchive 类进行的。因此，关于 CSocketFile 类的其他函数请参看 MSDN，这里不再进行赘述。

4.3.2　使用 CArchive 类进行串行化

在 MFC 中，CArchive 对象可以将数据序列化（按照顺序）写入与它相关联的文件中去。它提供类型安全的缓冲机制。下面将讲一下 CArchive 类常用的函数。

1．工作原理

CArchive 类对象在初始化时，先指定一个缓冲区作为临时存储，再将需要保存的数据写到缓冲区中。当缓冲区被填满时，才将缓冲区中的内容写入它所指向的 CFile 文件对象中。

同样，当用户读取数据时，串行化对象将数据从文件读取到指定的缓冲区，再从缓冲区读取到与对象相关联的文件中。这样，使用缓冲区不但减少了对物理硬盘的操作次数，而且提高程序的运行速度。

2．串行化对象

一般，CArchive 类使用构造函数创建指定的串行化对象，并且与 CSocketFile 对象相关联。原型如下：

```
CArchive::CArchive(CFile *pfile, UINT nMode, int nBufsize, Void *lpBuf=
NULL);
```

参数 pfile 指向一个需要进行串行化的对象指针。nMode 是设置创建对象的标志。如果用户设置了此标志，则必须在对象销毁前调用 Close()函数关闭文件。否则，文件中的数据将会被损坏。该参数的常用标志如表 4.5 所示。

表 4.5　nMode的常用标志

常 用 标 志	标志所示意义
CArchive::load(store)	从文件中读取（保存）数据
CArchive::bNoFlushOnDelete	是为了防止 CArchive 对象在被销毁时候自动调用 Flush 进行更新

参数 nBufsize 用于设置的缓冲区大小；lpBuf 用于自定义缓冲区，默认情况下为 NULL。例如：

```
CSocket  *m_clientsocket=new CSocket;                    //创建套接字
```

```
CSocketFile *m_sockfile=new CSocketFile(&m_clientsocket);
                                         //创建与 m_clientsocket 关联的对象
CArchive *m_archive=new CArchive(&m_sockfile,CArchive::load| CArchive::
store,100,NULL);
```

在代码中，为创建的串行化对象 m_archive 设置一个大小为 100 的缓冲区。最后一个参数设为 NULL，表明缓冲区由系统决定。

3．串行化操作

在 CArchive 类中，是使用函数 ReadString()和 WriteString()进行 CSocketFile 文件的读写操作。函数原型如下：

```
UINT CArchive:: ReadString(CString str);
       void CArchive:: WriteString(CString str1);
```

这两个函数均包含一个字符串类型的参数。但是，其具体含义却不同，分别如下：

- ❑ str 表示将读取后保存的字符串数据。
- ❑ str1 表示将写入的字符串数据。

除了上面的方法以外，还可以使用串行化操作的基本方法。代码如下：

```
...                               //省略部分代码
m_archive<<str;                   //向串行化对象 m_archive 写入字符串 str
m_archive>>str1;                  //从串行化对象 m_archive 读出数据到 str1
m_archive->Close();               //关闭串行化对象 m_archive
```

在这里用户需要注意，在关闭串行化对象后，与其相关联的文件对象也会随之被关闭。函数 CArchive::Close()用于清除 CArchive 类创建时所指定的缓冲区，再关闭 CArchive 对象，并且将 CArchive 对象和与之相关联的 CSocketFile 对象进行分离。

如果用户需要马上将数据写入到串行化对象中，需要用到 Flush 函数。它主要用于将缓冲区中剩余的数据强制地写入 CArchive 对象所关联的文件中。代码如下：

```
...                               //省略部分代码
m_archive->WriteString(str+"\r\n");//调用 CArchive 类的 WriteString 发送命令
                                  //在这里也可以使用m_archive<<str<<"\r\n";
m_archive->Flush();               //强制将数据 str 写入到串行化对象中
m_archive->Close();               //关闭串行化对象
```

在程序中，如果没有调用函数 Flush()，那么真正将数据写入到物理磁盘是在调用函数 Close()关闭串行化对象以后。因此，一些重要的数据需要使用 Flush()函数立即写入文件，以防丢失。

4.3.3　获取 FTP 服务器文件信息

当用户编程时，需要获取 FTP 服务器文件的列表，以便查看文件的相关信息。本节将向用户讲述怎样获取 FTP 服务器文件的相关信息。

1．获取文件列表

一般情况下，FTP 文件列表信息是通过客户端和服务器端之间的数据通道获取。编程

中，用户可以向服务器发送 LIST 命令，服务器接收到该命令以后会向客户端返回 FTP 目录下的文件列表信息。需要用户注意，在 PORT 模式下传输数据，客户端需要向服务器提交本地 IP 地址和用于返回数据的端口号。

```
CSocket m_Client;                              //客户端套接字变量
CString m_host;                                //IP 地址字符串变量
UINT nport,port=111;                           //端口号
m_Client.GetSockName(m_host,nport);            //调用函数获得本地的 IP 地址
m_host.Format(m_host+",%d",port);              //格式化字符串
```

用户使用 PORT 命令可以向服务器发送端口号码。格式如：“PORT”+string。其中 string 表示已经格式化的 IP 和端口字符串。代码如下：

```
m_archive->WriteString("PORT "+ m_host+"\r\n");
                                   //调用 CArchive 类的 WriteString()函数发送
m_archive->Flush();
```

客户端发送端口之后，必须在该端口上进行监听，以便接受服务器的连接请求。用户需要注意，在服务器和客户端连接关闭以前，服务器均按照此次的 IP 和端口与客户端进行数据交换。监听代码如下：

```
m_Client.Create(111,SOCK_STREAM,NULL);         //创建在 111 端口上监听的套接字
m_Client.Listen();                             //进行监听
```

现在，用户可以向服务器发送 LIST 命令获取相关文件的信息。发送 LIST 命令的代码如下：

```
Try
{                       //尝试发送命令 LIST 到服务器，以获取文件列表
m_archive->WriteString("LIST "+"\r\n");
                        //调用 CArchive 类的 WriteString()函数发送 LIST 命令
m_archive->Flush();
}
 Catch(CException e)
 { MessageBox("发送关闭命令失败！"); }            //抛出错误并处理错误
```

当用户向服务器发送 LIST 命令后，服务器会向客户端提供的 IP 地址和端口号发出连接请求。所以，当客户端在指定端口上监听到连接请求后，应该对该连接进行处理。

一般情况下，用户将套接字设置为非阻塞模式，避免出现等待状态。当监听套接字检测到有连接请求到来时才响应，否则套接字一直处于监听状态。用户可以使用 AP 函数 Accept()来响应服务器的连接请求。代码如下：

```
#define WM_ACCEPT WM_USER+100                    //自定义消息，用于响应连接请求
afx_msg Void OnAccept(WPARAM wParam,LPARAM lParam)   //声明响应连接请求的函数
...                                              //省略部分代码
BEGIN_MESSAGE_MAP(CMyApp, CWinApp)
//{{AFX_MSG_MAP(CMyApp)
        // NOTE - the ClassWizard will add and remove mapping macros here.
        //    DO NOT EDIT what you see in these blocks of generated code!
//}}AFX_MSG
ON_COMMAND(ID_HELP, CWinApp::OnHelp)
ON_MESSAGE(WM_ACCEPT, OnAccept)                  //处理消息映射
END_MESSAGE_MAP()
```

首先，用户需要自定义消息 WM_ACCEPT 用于响应连接请求，然后添加消息映射将自定义消息和实现函数关联起来。在 Win32 API 中，WSAAsyncSelect()函数可以将套接字设置为非阻塞模式，其原型如下：

```
int WSAAsyncSelect(SOCKET s,HWND hWnd,int wMsg,long lEvent);
```

用户使用该函数时，应该包含 Windows Socket 的头文件和相应的库文件。代码如下：

```
#include <winsock2.h>                    //包含 Windows Socket 头文件
#pragma comment(lib,"WS2_32.lib")        //编译时连接 WS2_32 库
...
```

使用函数 WSAAsyncSelect()时，参数 wMsg 表示自定义消息 WM_ACCEPT。参数 lEvent 表示通知码，其取值如表 4.6 所示。该函数调用成功后，会一直检查通知码，直到指定的网络事件发生，否则将返回 0。

表 4.6　lEvent取值

取　　值	含　　义
FD_READ	表示套接字接收到对方发送的数据，用户可对其进行读取
FD_WRITE	通知用户可以继续发送数据
FD_ACCEPT	表示套接字上有连接请求到来
FD_CONNECT	套接字连接成功
FD_CLOSE	套接字检测到对应的连接被关闭

在这里，用户只检测有无连接请求到来，所以 lEvent 设置为 FD_ACCEPT。代码如下：

```
::WSAAsyncSelect(m_Client,m_hSocket,this->m_hWnd, WM_ACCEPT, FD_ACCEPT|
FD_READ);
                                         //将套接字设置为非阻塞模式
```

当有对应的套接字请求到来时，程序调用自定义消息响应函数进行处理。代码如下：

```
void OnAccept(WPARAM wParam,LPARAM lParam)
{
    SOCKET ss;
    sockaddr_in adder;
    char sz[1024]={0};                   //定义缓冲区
switch(LOWORD(lParam))                   //参数 lParam 的低字节表示通知码
{  case FD_ACCEPT;                       //检测到有连接请求到来
 ss=::Accept(m_Client.m_hSocket,adder,sizeof(adder));
                                         //接受到来的连接请求，返回一个临时套接字
     ...                                 //省略部分代码
  case FD_READ:
      ::recv(ss,&sz,1024,0);             //接收数据到缓冲区
...                                      //省略部分代码
}}
```

客户端检测到连接请求后，调用函数 recv()进行接收，并将数据保存在缓冲区 sz 中。关于接收到的列表信息如何进行显示等操作将在最后一节实例中讲述。在这里，用户需要注意函数 Accept()调用成功后会返回一个临时套接字的句柄。

2. 获取FTP文件属性

用户发送 LIST 命令以后，服务器返回信息中包含了文件的一些属性，包括时间、大

小等。服务器返回的每条信息都是以“\r\n”结束，在每条信息中以空格分开。

首先，用户需要对缓冲区 sz 中的数据进行解析得到一条完整的消息。代码如下：

```
char buf[100]={0};                    //用于保存临时数据
for(int i=0;i<1024;i++)               //循环解析消息数据以获得一条完整的信息
{
     if(sz[i]!="\")buf[i]==sz[i];     //取得的信息不是“\”，则保存到临时变量
   else
    { if(sz[i+1]=="r")MessageBox("成功解析一条消息！");
                                      //如果取得的是结束符号，则提示成功提取
}
}
```

上面的代码用于提取一条完整的信息，并将其保存在临时变量 buf 中。接下来，用户可以对提取到的信息再进行详细的解析，以便得到具体的文件属性。代码如下：

```
char ch="a";                          //初始化字符变量
CString str="";                       //定义字符串
int i=0,j=0;                          //定义循环变量
while(ch!=""&&i<1024)
{
   if(buf[i]!=""&&buf[i+1]==EOF)str+=(CString)buf[i];
                                      //如果不是空格则保存在字符串变量中
else
{
ch=buf[i+1];                          //如果是空格则移动到下一个字符
  i+=1;
  j+=1;
str="";                               //将字符串变量重新设置
}
switch(j)                             //根据变量 j 进行选择信息字符段
{
  case 1:
         MessageBox("文件最后一次保存的日期是：%c",str);
  case 2:
         MessageBox("文件最后一次保存的时间是：%c",str);
  case 3:
         MessageBox("文件的大小是：%c",atoi(str));
  case 4:
         MessageBox("文件的名称是：%c",str);
}
    }
```

上述代码实现的功能是对一条信息进行分析，得到文件准确的保存日期、时间和大小。用户需要了解在 Windows 下，FTP 服务器返回的信息格式，例如，08-23-12 10:06AM 16056 list.txt。该字符串第一段“08-23-12”表示文件保存的日期。第二段“10:06AM”表示保存时间。第三段“16056”表示文件大小。第四段“list.txt”表示文件名称。

关于获取 FTP 文件其他属性的方法，请用户参考其他书籍，本书将不再进行讲述。

4.3.4　上传文件

向 FTP 服务器上传文件的功能是当用户使用 FTP 客户端时，会经常使用到的一个功能。

若想实现该功能，上传文件的命令应该是 STOR 或者*STOU。STOR 命令会覆盖原有文件，而*STOU 命令则不覆盖已经存在的文件。当上传命令发送后，用户便可以直接传送文件。代码如下：

```
m_archive->WriteString("STOR "+"\r\n");
                               //调用 CArchive 类的 WriteString()函数发送 STOR 命令
char buff[1024]={0};           //设置缓冲区
SOCKET sock;                   //与服务器建立连接成功后返回的套接字句柄
CFile file("list.txt",CFile::modeReadWrite);
                               //关联文件并指定文件属性为可读可写
file.Read(buff,1024);          //读取文件内容到缓冲区中
file.close();                  //读取完毕，关闭文件
::Send(sock,buff,1024,NULL);   //调用 Send()函数发送文件内容到 FTP 文件
```

在代码中，用户必须先发送 STOR 命令到服务器，再将本地文件的内容读取到指定的缓冲区中，利用函数 Send()传送到服务器。这里利用 4.3.3 节中与服务器连接成功后返回的套接字进行传送数据。

注意：用户上传文件到 FTP 服务器时，如果不希望覆盖原有的文件，则应该将 STOR 命令改为*STOU 命令。

因为大部分文件的结束表示都是 EOF，所以当用户需要上传比较大的文件时，应该利用循环读取文件的方式进行编程。

4.3.5　下载文件

当用户从 FTP 服务器下载文件时，使用到的 FTP 命令是 RETR。该命令的用法与上传命令的用法相似。首先，客户端向服务器发送 RETR 命令，然后根据获取的文件大小，利用函数 Recv()进行接收。如果接收到的数据小于文件大小则继续接收，否则关闭数据连接即可。获取文件大小和文件名称的方法在 4.3.3 节中已经讲解，这里不再赘述。下载文件的代码如下：

```
int lenth;                                              //已经获取的文件大小
CString filename;                                       //已经获取的文件名称
int i=0;
m_archive->WriteString("RETR "+"\r\n");
                               //调用 CArchive 类的 WriteString()函数发送 RETR 命令
char buff[1024]={0};           //设置缓冲区
SOCKET sock;                   //与服务器建立连接成功后返回的套接字句柄
CFile file(filename,CFile::modeReadWrite); //建立文件并指定文件属性为可读可写
while(lenth!=0)
{
::Recv(sock,buff,1024,NULL);   //在套接字上接收数据到缓冲区中
file.Write(buff,1024);         //将缓冲区内容写到文件中
lenth=lenth-1024;              //从文件总大小中减去已经接收并写入文件中的大小
}
MessageBox("文件下载成功！");  //否则提示文件下载成功
```

在代码中，用户也可以使用获取到的文件大小设置接收缓冲区大小，但是这样做会导致一些不可预见的错误发生。

4.4　创建客户端

本节将用具体的 FTP 类和方法进行编程，使用户对 FTP 编程知识进一步巩固。在实例中，首先进行工程的创建、CFtp 类的封装，然后使用 CFtp 类进行编程应用。

4.4.1　建立工程

用户在 VC 中建立基于对话框的 FTP 客户端工程，可以直接通过向导创建。具体步骤如下：

（1）打开 VC，选择“文件”|“新建”命令，打开“新建”对话框，如图 4.2 所示。

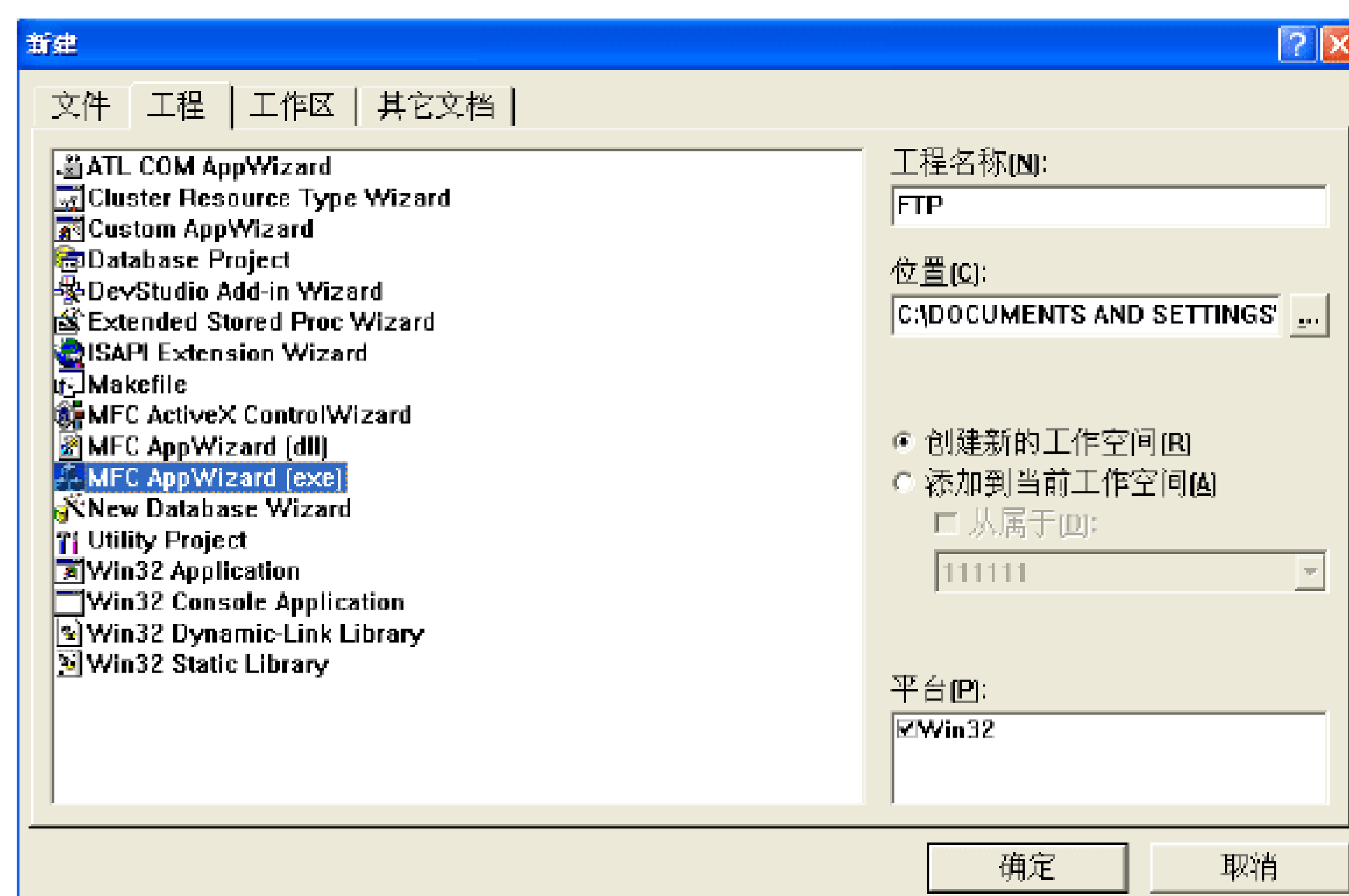


图 4.2　新建 MFC 工程

（2）在“工程”选项卡中，选择 MFC AppWizard[exe]项。在“工程名称”文本框中输入项目名称，本节实例名为 FTP，然后选择保存路径。单击“确定”按钮，进入如图 4.3 所示的界面。

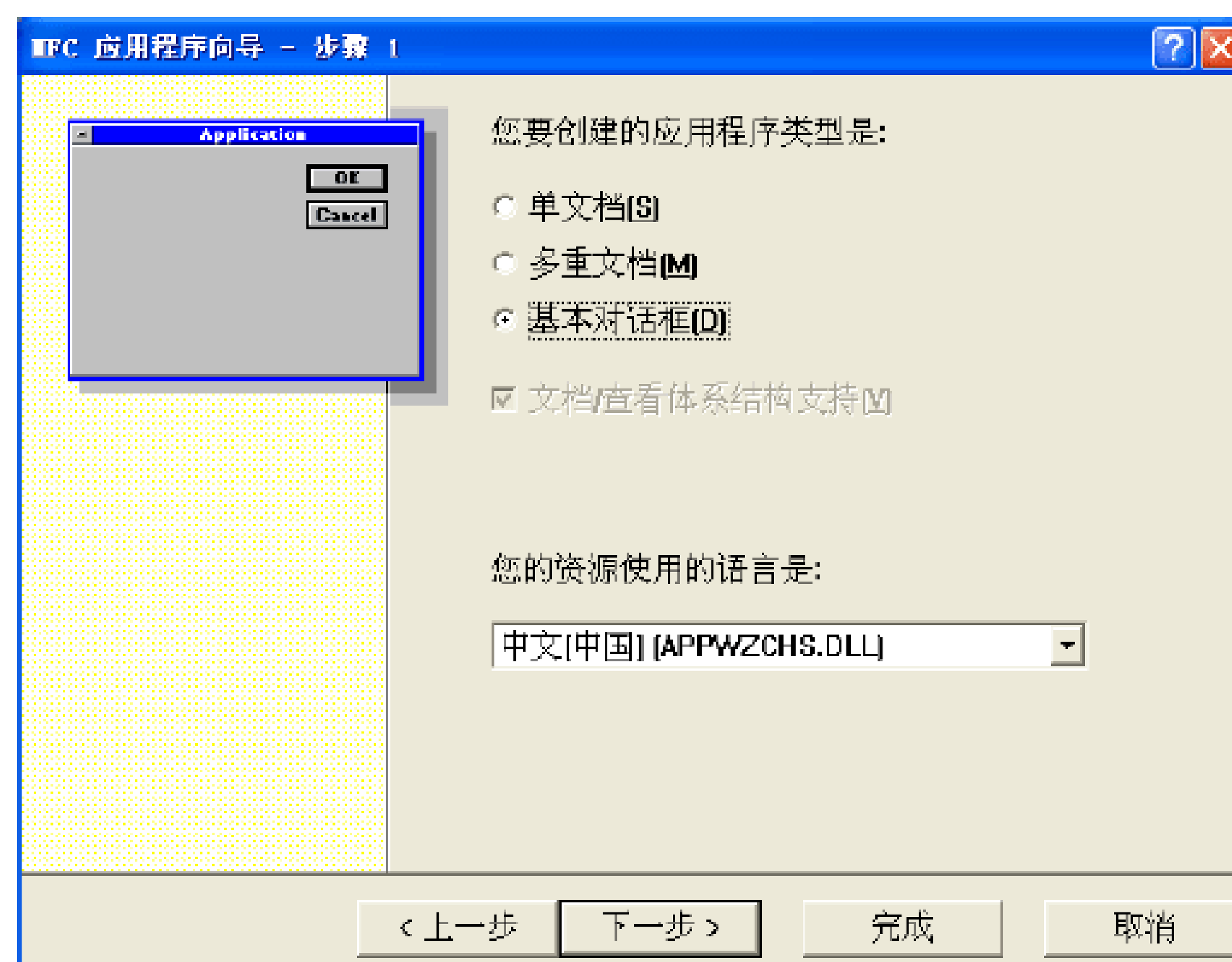


图 4.3　选择基于对话框项目

（3）在图 4.3 中，选择“基本对话框”单选按钮，其他的选项默认，单击“下一步”按钮，进入如图 4.4 所示的界面。

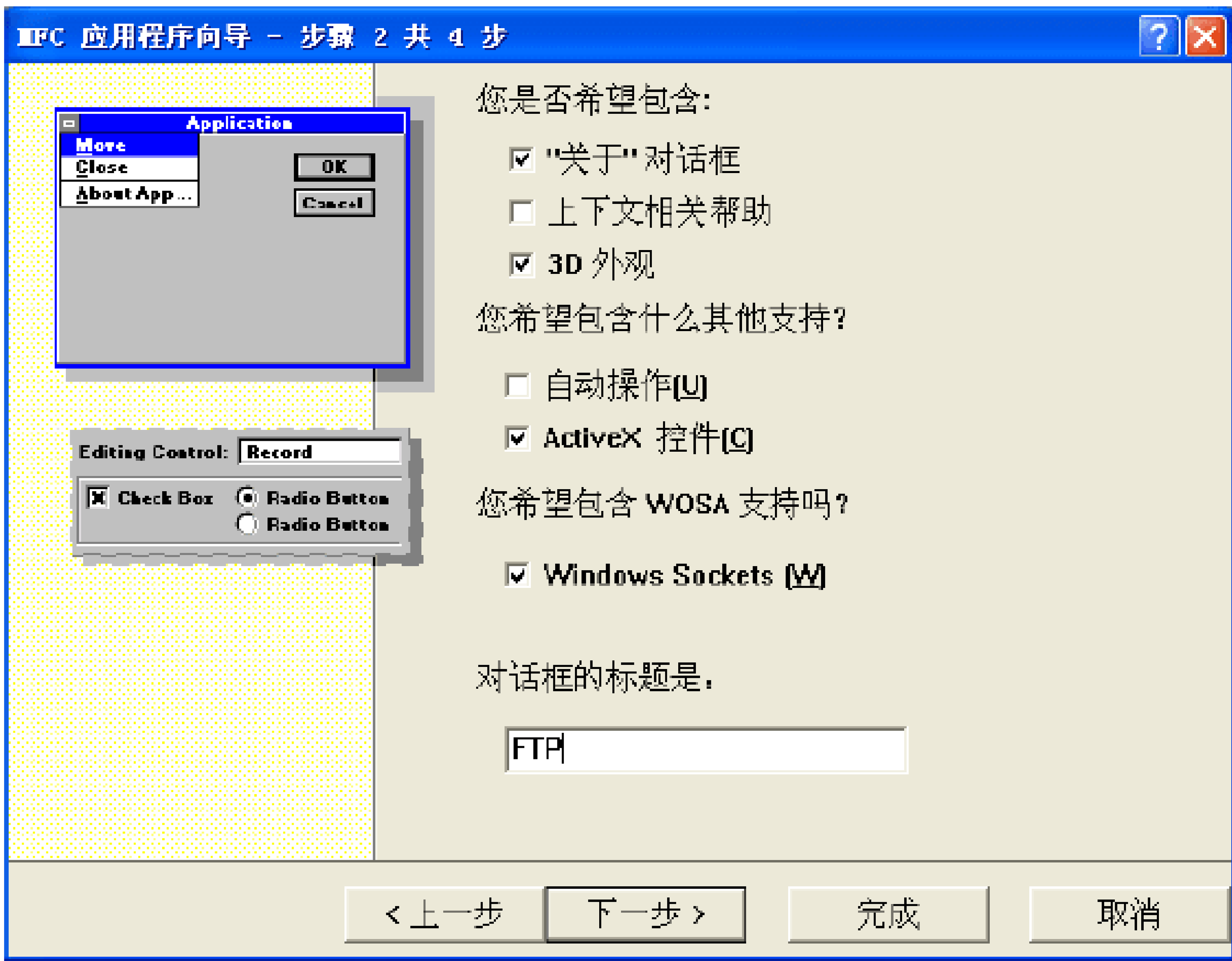


图 4.4　选择 Windows Sockets 项目

由于在实例程序中需要使用 WinSock，所以这里需要选择 Windows Sockets 复选框，其他选项默认。

（4）选择“下一步”按钮，进入如图 4.5 所示界面，直接单击 “完成”按钮即可完成项目的建立。

完成设置以后，将进入 VC 主界面。由于本项目基于对话框模式，所以用户需要拖动所需控件到对话框面板上即可，界面的最终效果如图 4.6 所示。关于添加消息映射等具体操作将在 4.4.3 节中进行讲解。到这一步，该实例工程已经建立完毕。

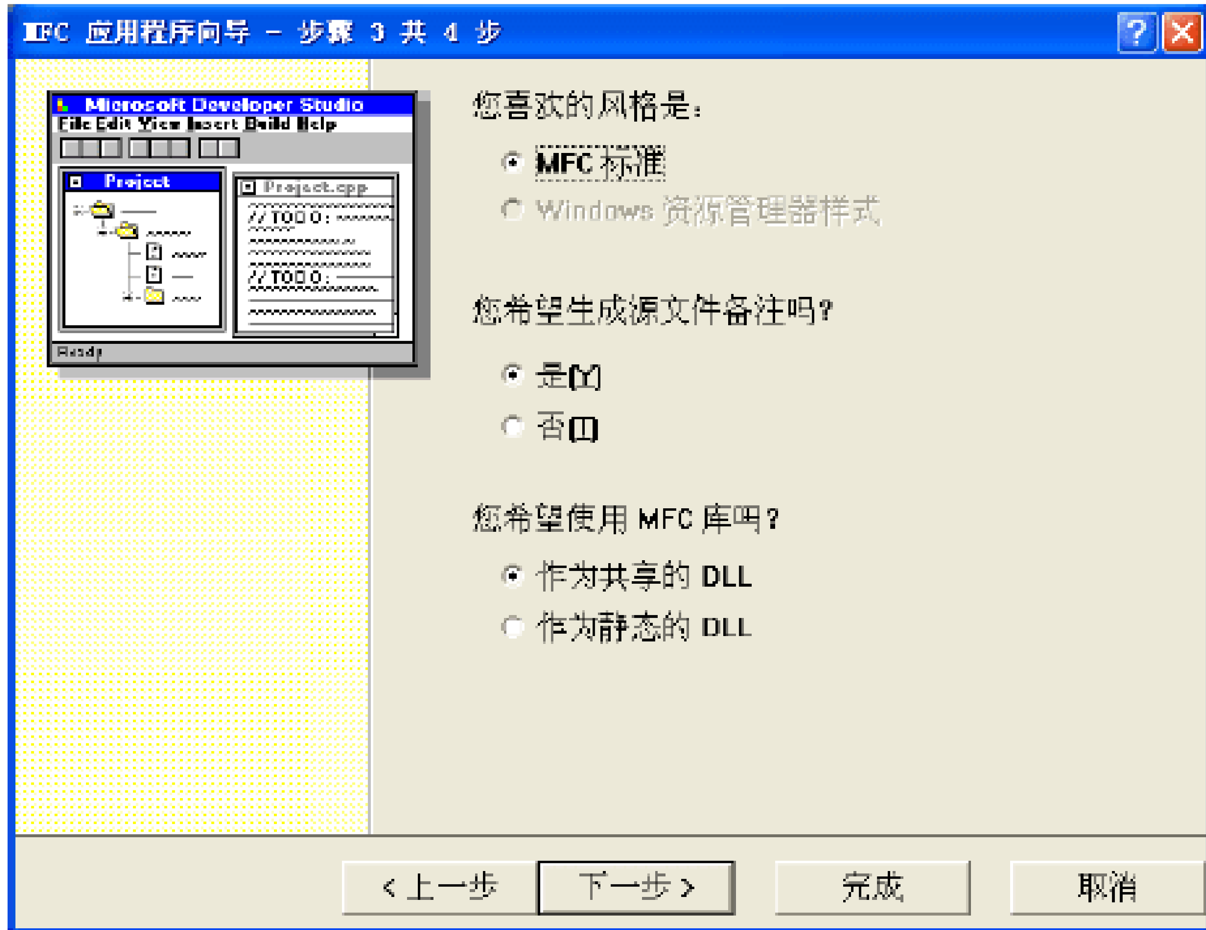


图 4.5　完成设置

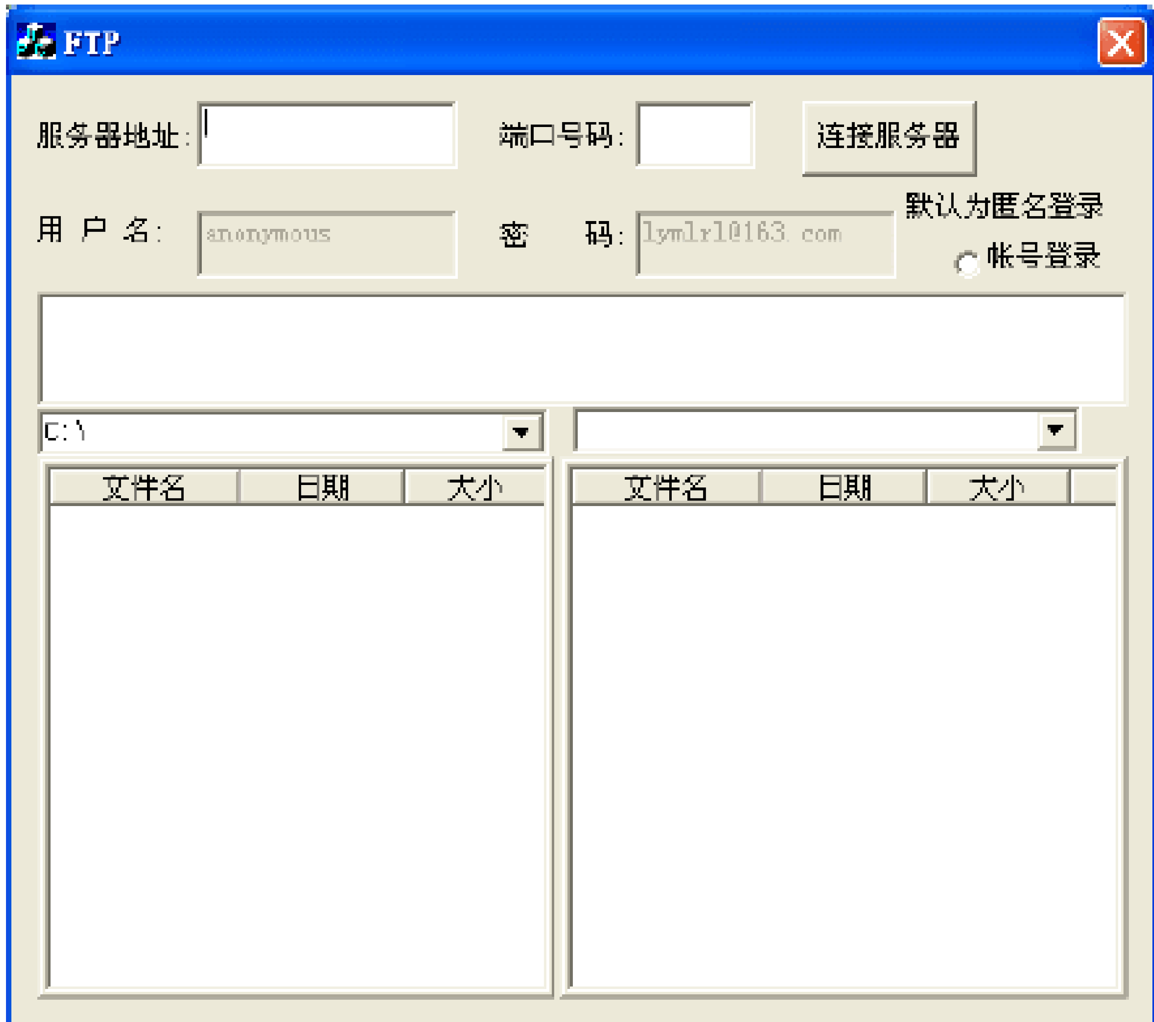


图 4.6　界面截图

在对话框中，用户首先输入服务器地址（端口号码默认为 21）、用户名和密码，然后单击“连接服务器”按钮连接指定 FTP 服务器。客户端与服务器的连接状态和用户操作等信息均会显示在中间的编辑框中。部分控件 ID 以及其表示内容如表 4.7 所示。

表 4.7　部分控件ID以及表示内容

内容	服务器地址	端口号码	用户名	密码	客户端状态	本地文件列表	服务器文件列表	连接按钮
ID	IDC_EDIT1	IDC_EDIT2	IDC_EDIT3	IDC_EDIT4	IDC_EDIT5	IDC_LIST1	IDC_LIST2	IDC_Connect

4.4.2　定义 CFtp 类

CFtp 类是用户自定义实现 FTP 功能和原理非常重要的类。在本节中，用户将学习到怎样在 VC 环境中定义和封装 CFtp 类。首先定义头文件 Ftp.h，代码如下：

```
class CFtp                                              //定义 CFtp 类
{
  public:
CString ipaddr,name,password;                           //IP 地址、用户名、密码
            int port;                                   //端口号码
        BOOL FTPConnect(CString severhost,int port);          //连接 FTP 服务器
        CSocket *m_clientsocke;                         //套接字对象
            CArchive *archive;                          //串行化对象
            CSocketFile *socketfile;                    //套接字文件对象
  private:
        CString Recv();                                 //接收命令消息
           Void Send(CString st);                       //发送命令消息
           Void UpdataFile(CString str);                //上传文件
           Void DownLoadFile(CString str1);             //下载文件
        Void GetFileStatu(char ch);                     //获取文件属性
}
```

从上面的 CFtp 类声明中，用户可以看到 FTP 编程相关的数据和实现方法。下面将根据 FTP 基本功能介绍每个函数。首先，客户端应该连接服务器，登录方式为匿名。其函数实现如下：

```
BOOL CFtp::FTPConnect(CString severhost, int port)
{
CSocket  *m_clientsocket=new CSocket();                 //构造连接套接字对象
m_clientsocket->Create(21,SOCK_STREAM,FD_READ|FD_WRITE,NULL);
                                                        //创建流式套接字
if(!m_clientsocket)                                     //判断套接字对象创建是否成功
{ MessageBox("套接字创建失败! ");
return false; }                                         //创建 m_clientsocket 失败
  if(!(m_clientsocket->Connect((atoi)severhost, port)))
return false;                                           //连接 FTP 服务器
   else
{ return true;}                                         //连接成功将返回 true
}
```

客户端连接 FTP 服务器，成功则返回 true，否则返回 false。如果连接成功，则需要向服务器发送命令以初始化服务器和获取服务器文件列表。函数 Send()定义如下：

```
Void CFtp::Send(CString charstring)                 // Send()函数发送信息到服务器
{ CSocketFile * socketfile;                         //定义对象指针
socketfile =new CSocketFile(m_clientsocket);
```

```
                                          //关联对象 m_clientsocket 是创建的套接字
archive=new CArchive(&m_sockfile, CArchive::load| CArchive::store);
                                          //创建对象 m_archive 的实例并指定属性
      charstring=""USER"+lymlrl+"PASS"+123456";     //构造字符串 charstring
      archive.WriteString(" "+"\r\n");        //向服务器发送空字符串进行初始化
 try{
archive ->WriteString(charstring +"\r\n");
                                          //调用 CArchive 类的 WriteString 发送命令
archive ->Flush();                        //强制写入命令到服务器
}
  Catch(CException e)                     //处理被抛出的异常
 {
 MessageBox("发送关闭命令失败! ");
}
}
```

当命令发送后，服务器会返回客户端请求的数据。函数 Recv()的实现如下：

```
                                              // Recv()函数接收服务器返回的数据
CString CFtp::Recv()
{
CString recvstr=" ";                          //初始化字符串 recvstr 为空
if(archive->ReadString(recvstr))              //接收返回信息并放到 recvstr 变量
{
if(recvstr==" ") MessageBox("接收数据为空");         //如果接收的数据为空则提示
{ MessageBox("接收数据成功");
return recvstr;                                    //返回数据
}}                                                 //返回接收到的数据
else
     {
MessageBox("接收数据失败");
}                                                  //提示接收数据失败
}
```

函数 Recv()利用“archive->ReadString(recvstr)”读取服务器返回的数据或者其他信息。其中，包括文件的属性等信息。用户可以从服务器返回的数据中读取文件的属性。函数实现如下：

```
Void CFtp::GetFileStatu(char car)             //参数 car 表示接收到的数据
{
   char buf[100]={0};                         //用于保存临时数据
   char ch="a";                               //初始化字符变量
CString str="";                               //定义字符串
int i=0,j=0;                                  //定义循环变量
for(int i=0;i<1024;i++)                       //循环解析消息数据以获得一条完整的信息
{
      if(car[i]!="\")buf[i]==car[i];          //取得的信息不是“\”，则保存到临时变量
    else
{
if(car[i+1]=="r")MessageBox("成功解析一条消息! ");
                                              //如果取得的是结束符号，则提示成功提取
}
}
while(ch!=""&&i<1024)
{
   if(buf[i]!=""&&buf[i+1]==EOF)str+=(CString)buf[i];
```

```
                                       //如果不是空格，则保存在字符串变量中
else
{
ch=buf[i+1];                           //如果是空格，则移动到下一个字符
  i+=1;
  j+=1;
str="";                                //将字符串变量重新设置
}
switch(j)                              //根据变量 j 进行选择信息字符段
{
  case 1:
        MessageBox("文件最后一次保存的日期是：%c",str);
                                       //打印文件各属性
  case 2:
        MessageBox("文件最后一次保存的时间是：%c",str);
  case 3:
           MessageBox("文件的大小是：%c",atoi(str));
  case 4:
        MessageBox("文件的名称是：%c",str);
}
 }
    }
```

函数 GetFileStatu()根据参数 car 所指向的接收内容数组，通过循环方式获取一条完整的信息，然后再从这条信息中取得各属性。

CFtp 类中很重要的作用是上传和下载文件，这两个功能的实现方法如下：

```
Void CFtp::UpdataFile(CString str)         //参数 str 表示上传文件的路径
{
archive->WriteString("STOR "+"\r\n");
                        //调用 CArchive 类的 WriteString()函数发送 STOR 命令
char buff[1024]={0};                //设置缓冲区
SOCKET sock;                        //与服务器建立连接成功后返回的套接字句柄
CFile file(str,CFile::modeReadWrite);  //关联文件对象并指定文件属性为可读可写
file.Read(buff,1024);                  //读取文件内容到缓冲区中
file.close();                          //读取完毕，关闭文件
::Send(sock,buff,1024,NULL);           //调用 Send()函数发送文件内容到 FTP 文件
}
```

函数 UpdataFile()根据参数 str 所指定的本地文件路径上传文件。首先读取本地文件内容到缓冲区中，利用函数 Send()将缓冲区的内容发送到服务器。下载文件函数 DownLoadFile()的实现方法与函数 UpdataFile()一样，其具体实现如下：

```
Void CFtp::DownLoadFile(CString filename)
                               //参数 filename 表示从列表中获取的文件名
{
int lenth;                     //已经获取的文件大小
int i=0;
archive->WriteString("RETR "+"\r\n");
                          //调用 CArchive 类的 WriteString()函数发送 RETR 命令
archive->WriteString(filename +"\r\n");    //向服务器发送将要下载的文件名称
char buff[1024]={0};           //设置缓冲区
SOCKET sock;                   //与服务器建立连接成功后返回的套接字句柄
CFile file(filename,CFile::modeReadWrite); //建立文件并指定文件属性为可读可写
while(lenth!=0)
```

```
{
::Recv(sock,buff,1024,NULL);    //在套接字上接收数据到缓冲区中
file.Write(buff,1024);          //将缓冲区内容写到文件中
lenth=lenth-1024;               //从文件总大小中减去已经接收并写入文件中的大小
}
```

本节封装了 CFtp 类发送命令、接收数据、获取文件属性和上传下载文件等 FTP 主要操作。用户在程序中使用该类时，应将其头文件“Ftp.h”和“Ftp.cpp”加入到工程中，然后创建 CFtp 类对象对各函数进行调用即可。

4.4.3　使用 CFtp 类编程

如图 4.6 所示，用户登录服务器的方式可以是程序默认的匿名登录，同时也可以使用指定账号登录。其响应代码如下：

```
void CFTPDlg::OnRadio2()
{
this->GetDlgItem(IDC_EDIT3)->EnableWindow(true);   //禁用 IDC_EDIT3 编辑框
    this->GetDlgItem(IDC EDIT4)->EnableWindow(true);
                                                    //禁用 IDC_EDIT4 编辑框
}
```

当用户输入服务器 IP 地址等信息后，单击“连接服务器”按钮，程序根据用户提供的信息对服务器进行连接。该按钮对应的消息响应函数代码如下：

```
void CFTPDlg::OnConnect()
{
   CString str,str1;                       //定义字符串变量
int port=0;                                //定义端口变量
this->GetDlgItem(IDC_EDIT1)->GetWindowText(str);       //获取 IP 字符串
this->GetDlgItem(IDC_EDIT2)->GetWindowText(str1);      //获取端口号码
port=(int)atoi(str1);                      //将端口字符串转换成数字
   if(ftp.FTPConnect(str,port))            //调用 CFtp 对象的函数进行连接
{
this->GetDlgItem(IDC_EDIT5)->SetWindowText("连接成功!"); //通知用户连接状态
 this->GetDlgItem(IDC Connect)->EnableWindow(false);
                                           //连接成功后，设置连接按钮为失效状态
ftp.Send("LIST/r/n");                      //发送命令获取文件列表信息
str=ftp. Recv();                           //接收数据
ftp.GetFileStatu(str.GetAt(0));            //获取文件名称
}}
```

在客户端启动时，应该能获取到本地默认文件夹下的文件。在 VC 中，查找文件的 API 函数主要是 FindFirstFile()和 FindNextFile()函数。函数原型分别如下：

```
                                                //开始查找文件并获得其句柄
HANDLE FindFirstFile(
  LPCTSTR lpFileName,
FINDEX_INFO_LEVELS fInfoLevelId,
LPVOID lpFindFileData,
FINDEX_SEARCH_OPS fSearchOp,
  LPVOID lpSearchFilter,
  DWORD dwAdditionalFlags
);
```

```
                                                  //从当前位置查找下一个文件
BOOL FindNextFile(
  HANDLE hFindFile,
  LPWIN32_FIND_DATA lpFindFileData
);
```

函数 FindFirstFile()可以在指定盘符下查找文件，并将获取到的文件数据保存到缓冲区中，该函数返回文件查找操作的句柄。参数 lpFileName 表示用户需要查找的文件名。如果该名称中没有包含路径，则程序会在当前目录下进行查找文件，否则在指定路径下查找文件。在文件名中可以使用“*”等通配符代替，如下：

```
lpFileName="C:\\windows\\*.*";      //在 C:\\windows\\下查找所有文件
lpFileName="C:\\windows\\*.txt";    //在目录 C:\\windows\\下查找所有 TXT 文件
lpFileName="C:\\windows\\vtk.bin"; //在目录 C:\\windows\\下查找文件 vtk.bin
lpFileName="C:\\windows\\*.exe";    //在目录 C:\\windows\\下查找所有 EXE 文件
```

函数 FindNextFile()可以继续查找其他格式的文件操作。参数 hFindFile 表示函数 FindFirstFile()返回的操作句柄。参数 lpFindFileData 指向结构体 WIN32_FIND_DATA，保存了程序所找到的文件名和文件属性等数据。该函数调用成功返回 true，否则，返回 false。WIN32_FIND_DATA 结构如下：

```
typedef struct  WIN32 FIND DATA {
    DWORD dwFileAttributes;              //文件属性
    FILETIME ftCreationTime;             //文件创建日期
    FILETIME ftLastAccessTime;           //文件最后保存日期
    FILETIME ftLastWriteTime;            //文件最后修改日期
    DWORD    nFileSizeHigh;              //文件长度的高 32 位
    DWORD    nFileSizeLow;               //文件长度的低 32 位
    DWORD    dwReserved0;                //现在保留
    DWORD    dwReserved1;                //现在保留
    TCHAR cFileName[MAX-PATH];           //本次查找到的文件名
    TCHAR cAlternateFileName[14];        //文件的短文件名
} WIN32_FIND_DATA;
```

注意：参数 cAlternateFileName[14]表示文件的短文件名。例如，文件路径为 C:\\windows\\vtk.bin 的文件短名称为 vtk.bin。

用户在客户端启动时，可以使用上面两个函数进行文件的查找。其代码如下：

```
BOOL CFTPDlg::OnInitDialog()
{
...                                               //省略部分代码
int i=0;
LVITEM item={0};                                  //初始化列表结构
item.mask=LVIF_TEXT;                              //指定 pszText 域有效
WIN32_FIND_DATA filedata={0};                     //初始化结构体 WIN32_FIND_DATA
HANDLE filehand;                                  //文件句柄
filehand=::FindFirstFile("C:\\*",&filedata); //查找 C 盘下所有文件
while(::FindNextFile(filehand, &filedata))
{
item.pszText=(LPTSTR) filedata. cFileName; //将文件名称赋给列表项
  this->GetDlgItem(IDC_LIST1)-> InsertColumn(i,& item); //在列表中插入栏目名称
```

```
i+=1;
}
return TRUE;
}
```

除了上述获取文件的方式以外，还可以通过用户选择的特定盘符进行获取。其中，响应用户选择的函数是 CFTPDlg::OnSelchangeCombo1()，代码如下：

```
void CFTPDlg::OnSelchangeCombo1()               //组合框选择消息响应
{
   CString str;                                 //定义字符串变量
 int i=m_c1.GetCurSel();                        //获取用户单击位置的索引
m c1.GetLBText(i,str);                          //获取索引处的字符
str+=" * ";                                     //添加通配符“*”
WIN32_FIND_DATA filedata={0};                   //初始化结构体 WIN32_FIND_DATA
HANDLE filehand;                                //文件句柄
filehand=::FindFirstFile(str,&filedata);        //查找特定盘下所有文件
while(::FindNextFile(filehand, &filedata))
{
item.pszText=(LPTSTR) filedata. cFileName; //将文件名称赋给列表项
this->GetDlgItem(IDC_LIST1)-> insertColumn(i,& item); //在列表中插入栏目名称
i+=1;
}   }
```

m_c1 是 CComboBox 类的对象。通过代码，程序可以获取用户所选择目录下的所有文件并且显示在列表中。用户获取服务器文件名称和获取本地文件名称的实现方法一样，使用 CFtp 类函数 GetFileStatu()获取服务器文件名称，然后设置列表控件的栏目即可，所以这里不再赘述。

在本地文件列表中，用户需要响应右键消息。在右键消息响应函数中获取文件名称，调用 CFtp 类的函数 UpDataFile()上传文件。

```
void CFTPDlg::OnRclickList1(NMHDR* pNMHDR, LRESULT* pResult)
{
CString str1;
int i= this->GetDlgItem(IDC_LIST1)->GetCurSel();   //获得单击鼠标位置的索引
CString str= this->GetDlgItem(IDC_LIST1)->GetText(i);
                                                   //获取索引位置的文件名称
WIN32_FIND_DATA filedata={0};                   //初始化结构体 WIN32_FIND_DATA
HANDLE filehand;
filehand=::FindFirstFile("C:\\*",&filedata);    //查找 C 盘下所有文件
while(::FindNextFile(filehand, &filedata))//在文件中查找与指定文件名称相同的文件
{
if(str==(LPTSTR) filedata. cFileName)
{
str1+="C:\\"+str;                               //构造文件完整路径
 ftp. UpdataFile("str");                        //上传指定文件
}
}}
```

上传函数中，使用列表控件中的函数 GetCurSel()获取指定索引，再调用函数 GetText()获取文件名称。然后使用函数 FindFirstFile()和 FindNextFile()查找对应文件，构造完整路径后调用 CFtp 类函数 UpDataFile()上传该文件。

在服务器文件列表中，响应右键消息。其消息响应函数如下：

```
void CFTPDlg::OnRclickList2(NMHDR* pNMHDR, LRESULT* pResult)
{
int i= this->GetDlgItem(IDC_LIST1)->GetCurSel();   //获得单击鼠标位置的索引
CString str= this->GetDlgItem(IDC_LIST1)->GetText(i);
                                                   //获取索引位置的文件名称
WIN32_FIND_DATA filedata={0};           //初始化结构体 WIN32_FIND_DATA
HANDLE filehand;
filehand=::FindFirstFile("ftp://127.0.0.1/ftp",&filedata);
                                        //查找服务器下 ftp 文件夹中内容
while(::FindNextFile(filehand, &filedata))
                                        //在文件中查找与指定文件名称相同的文件
{
 if(str==(LPTSTR) filedata. cFileName)
{
str1+=" ftp://127.0.0.1\ftp\"+str;     //构造文件完整路径
ftp.DownLoadFile(str);           //调用 CFtp 类的 DownLoadFile()函数进行下载
}
```

本节主要讲解了自定义类 CFtp 的使用方法，其用法非常简单。如果用户需要扩展其内容，首先在文件“Ftp.h”中自定义函数或数据。然后在“Ftp.CPP”中写出自定义函数的代码即可。本节中详细代码请参考光盘中 4.4.3 小节文件夹。

4.5　小　　结

本章主要向用户讲解了 FTP 的原理和各功能的实现方法。通过连接服务器、发送验证信息、发送命令、接收数据和上传下载文件等 FTP 常用操作方法进行程序编写，并封装了自定义类 CFtp 类。用户可以通过自定义类中的方法进行编程。通过本章相关知识的学习，用户应该掌握 FTP 编程的基本流程和方法。

第 5 章　网页浏览器

在如今网络流行的时代中，大多数用户都在使用微软的 IE 浏览器浏览网页。从以前的静态网页到现在丰富多彩的动态网页，用户都是通过网页浏览器进行浏览。网页浏览器应该具有解析 HTML 代码或者其他语言（如 ASP.NET 等）网页的功能。本章将向用户介绍浏览器的工作原理以及设计流程等知识。

5.1　HTTP 请求

通常情况下，设计过网页的用户都会知道客户端浏览器通过向服务器发送 HTTP 请求，服务器接受请求以后，将相应的网页内容传回客户端进行显示。这就是常见的 C/S（客户端/服务器）网络模型。客户端程序负责解析服务器传回的网页内容。

在 HTTP 中，请求就是客户端通过向服务器发送消息要求提供一定的服务的过程。请求方式有两种：GET 和 POST。

注意： C/S 模型是指网络通信的双方以特定角色进行数据传输。例如，从 IE 浏览器的角度来说，与网络服务器进行数据传输是基于 C/S 模型，浏览器相当于客户端；而从用户的角度来说，相当于是使用 IE 这个浏览器工具与服务器进行数据传输，所以该种方式是 B/S 网络模型。

5.1.1　GET 方式

GET 请求方式在网页设计中，被用来在客户端和服务器之间交换数据。该数据包括网页 HTML 内容、ZIP 或 RAR 等附件数据。当向服务器传送数据使用 GET 方式时，传送的数据会被显示在网络地址后面。例如，这个网址“http://218.6.132.5/luntan/?fromuid=539356”，所表示的内容是客户端首先将变量 fromuid 赋予值 539356，然后传送到服务器。

根据 GET 请求方式传送数据的特点，用户可以知道这种方式是不安全的。因为，用户所要传送的数据都会被显式地连接在网址后面，连接符号是“？”。但是，在邮箱中下载附件时所用的方式是 GET 方式。以 GET 方式向服务器传送数据的 HTML 代码如下：

```
<html>
        <head>
<title>GET 方式传送数据</title>
</head>
<body>
<form id=form1 name=form1 method="get" action="http://127.0.0.1/get.html">
```

```
<table border=0 cellPadding=1 cellSpacing=1 width=75%>
<tr><td width=150>姓名：</td>
      <td><input id=b1 name="name"></td></tr>
<tr><td width=150>地址：</td>
      <td><input id=b2 name="addres"></td></tr>
<tr><td width=150>电话号码：</td>
      <td><input id=b3 name="number"></td></tr>
<tr><td width=150>邮箱：</td>
      <td><input id=b4 name="email"></td></tr>
<tr><td><input type=submit value=保存>  <input type=reset value=
重置></td></tr>
</table>
</form>
</body>
</html>
```

代码在 IE 浏览器中运行的效果如图 5.1 所示。



图 5.1　代码运行效果

用户在表单中输入姓名、地址、电话号码和邮箱，单击“保存”按钮，浏览器会将数据赋予变量并连接在所提交的网络地址后面进行连接服务器。客户端根据用户所填内容构造的网络地址是：http://127.0.0.1/get.html/？name=liang&addres=zhongguo&number=0233564545&email=lymlrl@163.com。

用户需要注意，GET 方式会受到 URL 的最大长度限制，URL 的最大长度为 1024KB。所以，当用户需要向服务器传送较大数据时，应该选用 POST 方式进行传送。

5.1.2　POST 方式

与 GET 方式相反，POST 方式是隐式地进行数据传送。两者相比，POST 方式比较安全，因为用户所传送的数据不会被显示在网络地址后面，并且可以传送较大的数据，最大可以达到 2MB。

使用 POST 方式向服务器提交的数据通过消息结构体进行传递。一般情况下，POST 方式被用来传递用户所提交的一些数据。POST 方式的 HTML 代码如下：

```
<html>
        <head>
<title>POST 方式传送数据</title>
</head>
<body>
<form id=form1 name=form1 method="post" action="http:
//127.0.0.1/get.html">
<table border=0 cellPadding=1 cellSpacing=1 width=75%>
<tr><td width=150>姓名：</td>
      <td><input id=b1 name="name"></td></tr>
<tr><td width=150>地址：</td>
      <td><input id=b2 name="addres"></td></tr>
<tr><td width=150>电话号码：</td>
      <td><input id=b3 name="number"></td></tr>
<tr><td width=150>邮箱：</td>
      <td><input id=b4 name="email"></td></tr>
<tr><td><input type=submit value=保存>  <input type=reset value=
重置></td></tr>
</table>
</form>
</body>
</html>
```

代码运行后的界面与 GET 方式相同。当用户单击“保存”按钮以后，客户端连接服务器。同时将用户所填写的表单内容作为消息体加入到请求消息中，并且发送请求消息到服务器。

5.1.3　请求消息

请求消息是客户端为了获取服务器上的资源而向服务器发送的消息。该消息结构通常分为消息头和消息体，如上面所讲到的 POST 方式传递数据时，就会用到消息体。下面是一个缺少消息体的请求消息：

```
GET /FTP.html HTTP/1.1
Host：127.0.0.1
Accept：*/*
Referer：http://127.0.0.1/FTP1.html
Connection：close
```

上面的代码是基于 GET 方式的，Host 标题字段表示服务器的主机地址。该代码是请求服务器返回相对主机地址为/FTP.html 的 HTML 文件。

使用 GET 方式向服务器传送数据的请求消息如下：

```
GET /FTP.html/? name=liang& addrea=panzhihua HTTP/1.1
Host：127.0.0.1
Accept：*/*
Referer：http://127.0.0.1/FTP1.html
Connection：close
```

客户端向服务器传送数据，相当于向 FTP.html 页面传递参数。参数之间可以使用符号“&”连接。用户利用该特点可以不用打开邮箱而下载邮箱中的附件，只需要改变传入网页的参数即可。

用户在请求消息中，可以使用不同的标题字段描述请求的附加信息或者客户端信息。常见的消息标题字段如表 5.1 所示。

表 5.1　常见消息标题字段

标 题 字 段	意　　义
Accept	客户端希望接收的媒体类型
Accept-Encoding	客户端希望接收的数据的编码方式
Accept-Charset	客户端希望接收的数据的字符集
Form	指明客户端所提供的邮箱地址
Authorization	客户端向服务器提供身份验证的字段
Range	用于要求服务器返回部分数据的字段
Referer	记录客户端获得资源的URL地址
User-Agent	指明客户端身份的字段

5.2　HTTP 响应

HTTP 响应是指服务器对客户端的请求作出的反应，服务器的响应也是通过消息实现的。与请求消息一样，响应消息也是分消息头和消息体两部分组成，但是两者之间需要使用一个空白行分开。在消息头中包含了响应的当前状态和服务器的一些信息，消息体中则包含了响应的实体数据。如：

```
HTTP/1.1 200 OK
Date: Mon,21 Nov 2008 18:33:22 GMT
Sever: Microsoft-IIS/6.0
Accept-Ranges: bytes
Content-Type: image/bmp
Connection:close
                                    //使用空白行隔开
（响应的二进制实体数据）            //实体数据
```

消息的第一行为响应的当前状态信息，后面接着是响应的标题字段信息，空白行后的响应的实体数据。下面将向用户详细讲解各个信息的内容以及表示的含义。

5.2.1　响应状态信息

响应的状态信息包含在响应消息的第一行，由 HTTP 版本代号、响应码和响应状态描述文本组成。其中，响应码表示客户端此次请求是否成功或其他原因出错。用户可以从响应码中知道具体出错的原因，常见的一些响应码类别，如表 5.2 所示。

表 5.2　部分常见的响应码类别

响　应　码	分　　类	意　　义
200～299	成功	表示请求已经被服务器成功接收、理解
300～399	重定向	表示客户端需要根据服务器返回的信息作进一步请求
400～499	客户端出错	客户端的请求不能被服务器理解或满足
500～599	服务器出错	表示服务器不能满足或完成客户端的请求

表 5.2 中仅给出一部分常见的响应码类别，如果用户需要了解响应的具体情况，请参考 RFC2068，其中给出了具体响应码的含义。例如，响应码的一些具体含义，如表 5.3 所示。

表 5.3　响应码的一些具体含义

响　应　码	意　　义
201	服务器创建了一个新资源
202	服务器收到请求，但未处理完毕
204	请求成功，但返回空数据
300	返回多个请求结果，供客户端选择
301	请求的资源已经移动到新的永久URL上
302	请求资源被移动到一个临时URL上
304	请求的资源没有进行更新
400	出现请求错误
401	需要认证，而请求没有进行认证
403	服务器接收请求但不能访问请求资源
404	没有找到所请求的资源
405	服务器不允许该请求方式
501	服务器还没有实现请求的方法
502	网络的网关出现错误
503	服务器忙

如果用户在实际编程时，需要知道响应的具体状态信息可以对响应消息进行读操作，然后分离出响应码即可。在 RFC2068 中，对一些扩展的响应码没有作出相应的解释。这种情况可以简单地认为该响应码等于该类首个响应码的解释。例如，响应码 333（扩展的编码）在 RFC2068 中没有相应的解释，可以认为 333 等价于 300 的响应码解释，表示返回多个请求结果供客户端选择。

5.2.2　响应标题字段信息

在响应标题字段信息中包含了服务器返回除响应行以外的其他信息。

1．Location标题

当服务器上的资源被保存到其他地址以后，服务器会将新地址返回到客户端，这时在响应标题字段中会添加 Location 标题。该标题表示资源的实际位置，并且是绝对的 URL 地址。

```
HTTP/1.1 302 OK
Date: Mon,21 Nov 2008 18:33:22 GMT
Sever: Microsoft-IIS/6.0
Accept-Ranges: bytes
Content-Type: image/bmp
Location: http://127.0.0.1/mp3/20080632.wma           //指向一个绝对地址
Connection:close
```

通常情况下，该标题与响应码 302 一起使用，表示客户端所请求的资源已经被转到服务器的另外一个 URL 上。

2．Server标题

该响应标题表示服务器使用的软件名称和版本信息。例如：

```
Sever: Microsoft-IIS/6.0
```

Server 标题标识了服务器端 IIS 软件的版本号。

5.2.3　实体标题字段信息

在服务器的响应消息中含有实体数据，这些数据由实体标题进行描述。

1．Content-type标题

该标题可以用于指示实体数据的格式，以及所使用的字符集。

```
Content-type: text/html;charset=ASCII
```

上述字段的意思是实体数据是文本格式的 HTML 文件，所使用的字符集为 XLM。如果服务器返回一幅“XLM”或其他格式的图片到客户端，则该字段形式应如下：

```
Content-type: image/jpg
Content-type: image/bmp
```

2．Content-Length标题

该标题必须与 Content-type 标题一起使用，用于表示实体数据的大小（以字节为单位）。其用法如下：

```
HTTP/1.1 200 OK
Date: Mon,21 Nov 2008 18:33:22 GMT
Sever: Microsoft-IIS/6.0
Accept-Ranges: bytes
Content-Type: image/bmp
Content-Length: 1024
Connection:close
```

上述字段表示在服务器的响应消息中，实体数据是一幅 bmp 格式的位图，其大小为 1024B。关于一些不常用的实体标题，如 Content-Language、Last-Modified、Content-Base 等标题的用法，请读者自行参考其他相关资料，本书不再赘述。

5.2.4　实体数据

前面已经提到，在服务器的响应消息中包括了消息头和消息体两部分。其中，消息体中包含了实体数据，并且在消息头和实体数据之间使用一个空白行进行分隔。例如，客户端向服务器请求一个页面 GET.html，服务器的响应消息格式如下：

```
HTTP/1.1 200 OK              //消息头
Date: Mon,21 Nov 2008 18:33:22 GMT
Sever: Microsoft-IIS/6.0
Accept-Ranges: bytes
Content-Type: text/html
Content-Length: 1024
Connection: close
                             //用空白行进行分隔
<html>                       //消息体数据
        <head>
<title>GET 方式传送数据</title>
</head>
<body>
<form id=form1 name=form1 method="get" action="http://127.0.0.1/get.html">
<table border=0 cellPadding=1 cellSpacing=1 width=75%>
<tr><td width=150>姓名: </td>
      <td><input id=b1 name="name"></td></tr>
<tr><td width=150>地址: </td>
      <td><input id=b2 name="addres"></td></tr>
<tr><td width=150>电话号码: </td>
      <td><input id=b3 name="number"></td></tr>
<tr><td width=150>邮箱: </td>
      <td><input id=b4 name="email"></td></tr>
<tr><td><input type=submit value=保存>  <input type=reset value=
重置></td></tr>
</table>
</form>
</body>
</html>
```

在上面的响应消息中，服务器向客户端返回的响应消息中，响应码 200 表示请求被服务器理解并接收。返回的实体数据是一个网页内容，其格式为*.html 格式，大小为 1024B。

总之，服务器返回的响应消息类似于 C++语言中的结构体，消息头和消息体就是这个结构体里面的元素。用户在使用 HTTP 编程时，可以根据需要自定义一个结构体存储该消息数据。例如，自定义一个简单的消息结构体。

```
typdef struct
{
   char *messagehead;            //数据头指针
float i;                         //实体数据的大小
   char *messagebody;            //实体数据指针
...
} message;
```

这个结构体的用法很简单，例如利用该类获取响应消息的响应码，代码如下：

```
...
message msg;                    //结构体对象
CString str;                    //存放响应码
msg. messagehead=&recvdata;     // recvdata 为接收到的响应消息
for(int i=9;i<=11;i++)          //响应码位于数据头的第九位
{
str+= msg. messagehead+i;       //将获得的响应码存放于 str 中
  }
int j=::atoi(str);              //将 str 转换为整型变量
```

```
str.Format("消息响应码为: %d\n",j);   //格式化字符串
MessageBox(str);                      //输出格式化字符串，通知用户消息响应码
```

由于消息响应码位于数据头的第九位到第十一位，所以在代码中直接使用了响应码准确位置进行查找。如果用户在预先不知道的情况下，则必须利用指针进行移位查找。当然也可以使用 CString 类进行查找，也就是将常用的一些响应码存入文件中，然后使用函数 CString::Find()与文件中的数据进行比较查找亦可。实现的方法有很多，具体方法视用户而定。

5.3　制作个性化界面

在 VC 中制作与 IE 功能相似的网页浏览器，可以使用 MFC 中的 CHtmlView 类，也可以使用 ActiveX 控件类 CWebBrowser2 实现网页浏览器的开发。对于这两种方法，本章将分别进行讲解和实例编程。本节将向用户介绍怎样在 Visual C++ 6.0 中制作网页浏览器界面，包括工具栏等编程方法。

5.3.1　工具栏编程

对于网络浏览器而言，工具栏是很重要的一部分，在工程中使用工具栏可以方便用户的操作。用户通过工具栏上的地址栏输入网页地址，然后进行连接浏览。该工程中的工具栏应当包括浏览记录等功能按钮，地址栏和连接按钮则放置到另一对话框上。

1. 界面设计

在工程中添加一个对话框作为地址栏等控件的面板，ID 为 ID_DILOG，将组合框与连接按钮放置到对话框面板上，最终界面如图 5.2 所示。



图 5.2　界面效果

界面中各个控件 ID 以及属性如表 5.4 所示。

表 5.4　控件ID及其属性

控件ID	IDC_STATIC	IDC_COMBO1	IDC_BUTTON1
属性	地址	地址输入框	连接
用处	标识	用户输入地址	连接网络地址

界面设计好以后，用户需要为该对话框关联一个新类。在 VC 中，使用鼠标双击该对话框可以为其添加相应的类。双击后会弹出 Adding a class 对话框，如图 5.3 所示。选择 Create a new class 单选按钮，然后单击 OK 按钮，弹出 New Class 对话框，为新类命名和指定基类，如图 5.4 所示。

在本工程中，将对话框关联的类名设置为 CTooldlg，基类指定为 CDialog。

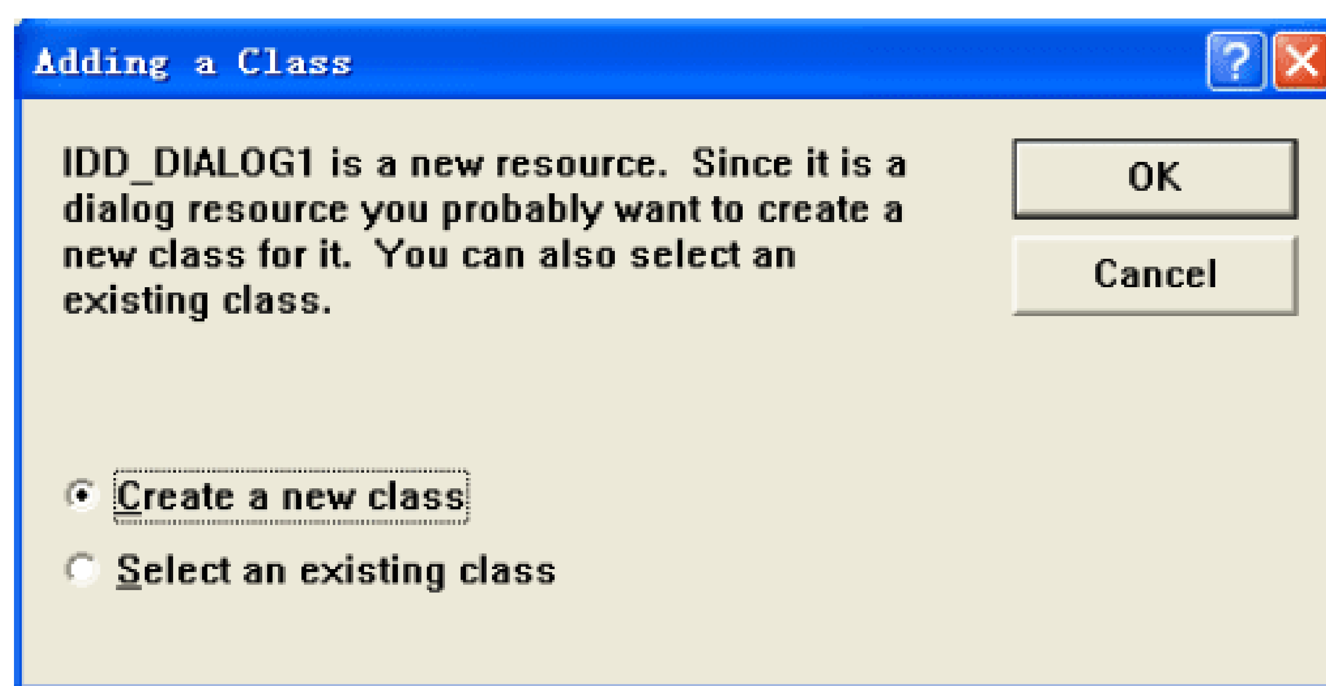


图 5.3 Adding a class 对话框

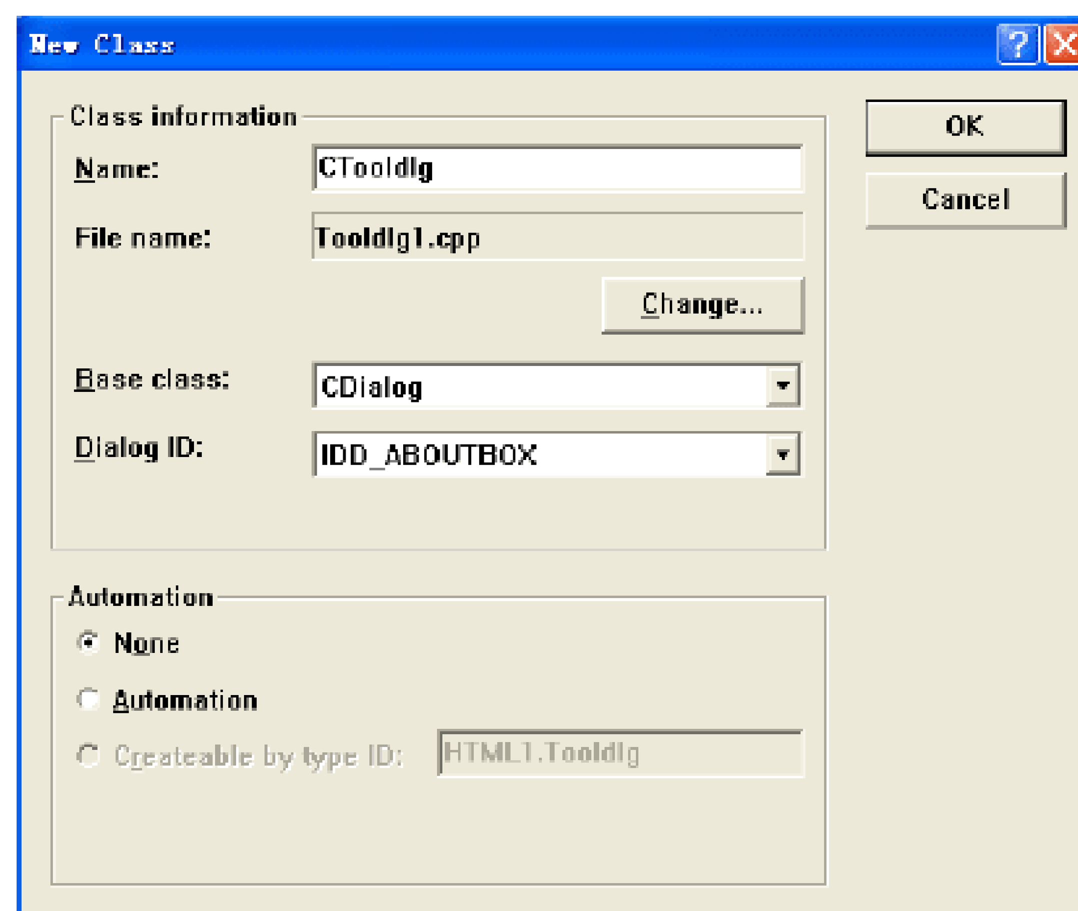


图 5.4 New class 对话框

2．添加对话框到工具栏

将 ID 为 ID_DILOG 的对话框添加到工具栏中，用户需要将 CTooldlg 类对象设置为 CMainFrame 类的成员变量。

首先，在 CMainFrame 类的头文件“MainFrm.h”开头处添加 CTooldlg 类的头文件“Tooldlg.h”。

```
...                                    //省略部分代码
#include "Tooldlg.h"                   //包含 CTooldlg 类的头文件
...                                    //省略代码
```

然后，在 CMainFrame 类中声明 CTooldlg 类对象 dlg，保护属性为 protected。

```
protected:
protected:
CStatusBar  m_wndStatusBar;            //声明状态栏对象
CToolBar    m_wndToolBar;              //工具栏对象
CReBar      m_wndReBar;                //CReBar 类对象
CTooldlg  dlg;                         //声明 CTooldlg 类对象 dlg
...                                    //省略部分代码
```

最后，在 CMainFrame::OnCreate()函数中创建 CTooldlg 类对象 dlg，并且将该对象添加到工具栏中。在这里，用户需要用到 MFC 中的 CReBar 类，该类相当于一个容器，可以将多个控件组合在一起。代码如下：

```
int CMainFrame::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
    ...                                //省略部分代码
if(dlg.m_hWnd==NULL)                   //判断 CTooldlg 类对象是否已经存在
{
    dlg.Create(ID_DILOG,this);         //创建 CTooldlg 类实例，并且与对话框
                                         ID DILOG 进行关联
    m_wndReBar.AddBar(&dlg);           //将实例对象 dlg 添加到容器中
}
    return 0;
}
```

通过上述代码，用户已经将带有地址栏的对话框 ID_DILOG 添加到工具栏中，效果如

图 5.5 所示。

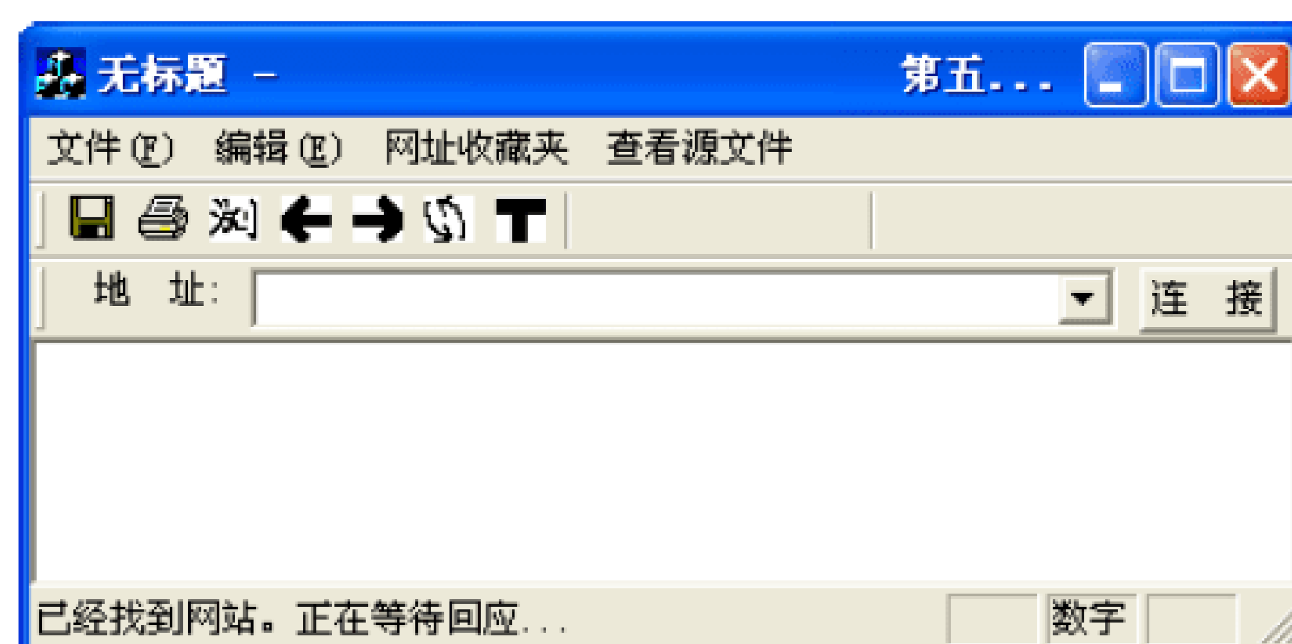


图 5.5　添加地址栏后的效果

用户使用程序浏览网页时，在地址栏中输入网址，然后单击“连接”按钮即可浏览相应的网页内容。注意：在本章中，如果没有实现相关按钮的消息响应函数，那么响应函数将在 5.3.2 节中进行实现。

3．添加工具栏按钮

当制作网页浏览器时，用户还需要添加一些功能。例如，刷新、上一步、下一步和浏览记录等。这些操作在 VC 中实现非常简单。

首先，在资源管理器中展开 Toolbar 项，添加 4 个工具栏按钮，如图 5.6 所示。各按钮的 ID 分别为 ID_VIEWRECORD、ID_PRE、ID_NEXT、ID_REFRUSH，分别表示浏览记录、上一步、下一步和刷新。

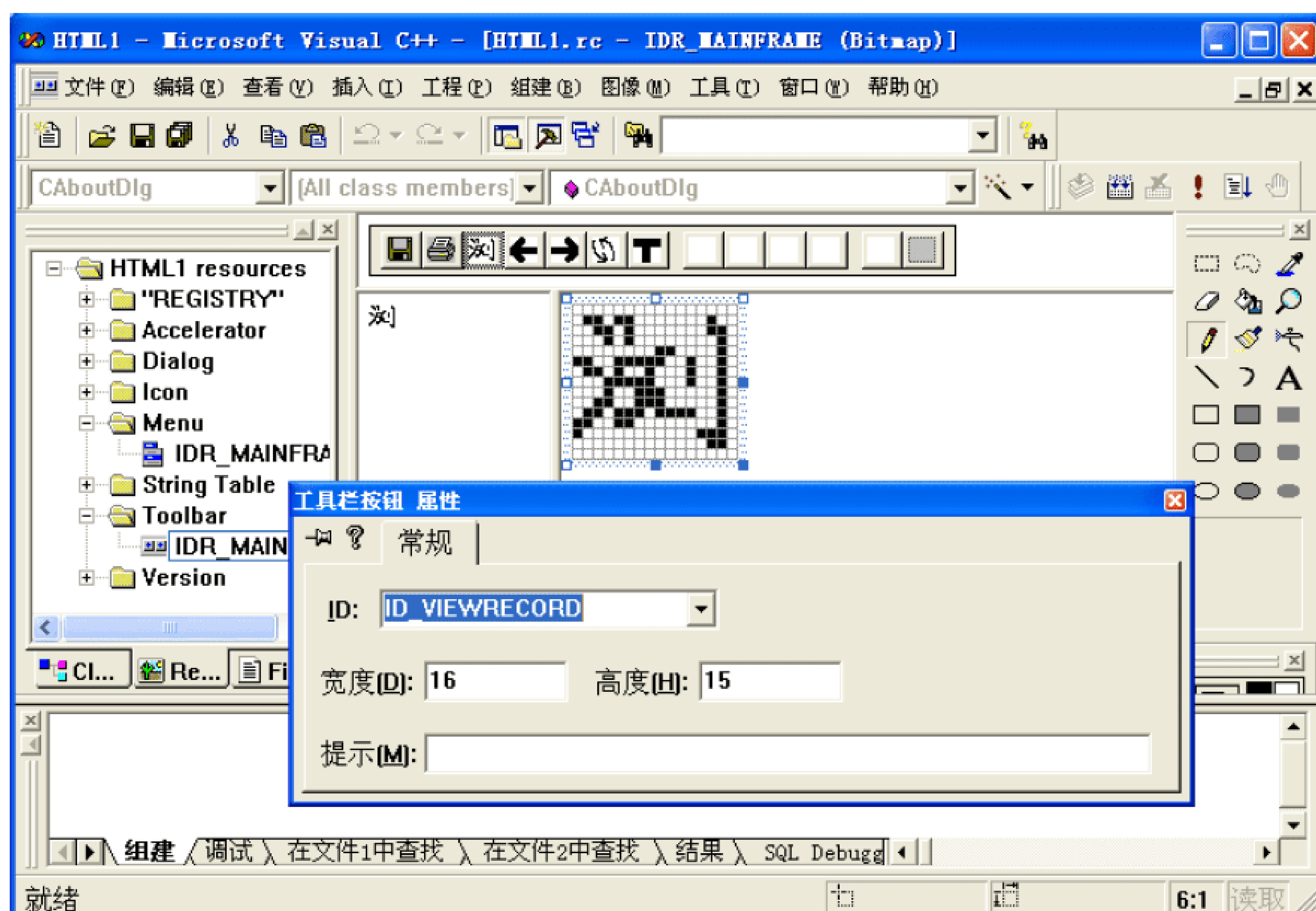


图 5.6　添加工具栏按钮

然后，编译执行该程序，程序效果如图 5.7 所示。



图 5.7　程序运行效果

用户会发现在运行的程序中，刚添加的工具栏按钮是灰色的。这是因为用户只是在工具栏中添加了按钮，并没有为其添加相应的消息响应函数。

5.3.2　添加消息响应

在 5.3.1 节中，用户添加了工具栏按钮和地址栏等控件，但是并没有为其添加消息响应函数。本节将讲述添加消息响应函数以及各按钮功能的具体实现方法。

首先，添加“连接”按钮的消息响应函数。在 VC 主界面中，按下 Ctrl+W 快捷键，弹出 MFC 向导对话框，如图 5.8 所示。

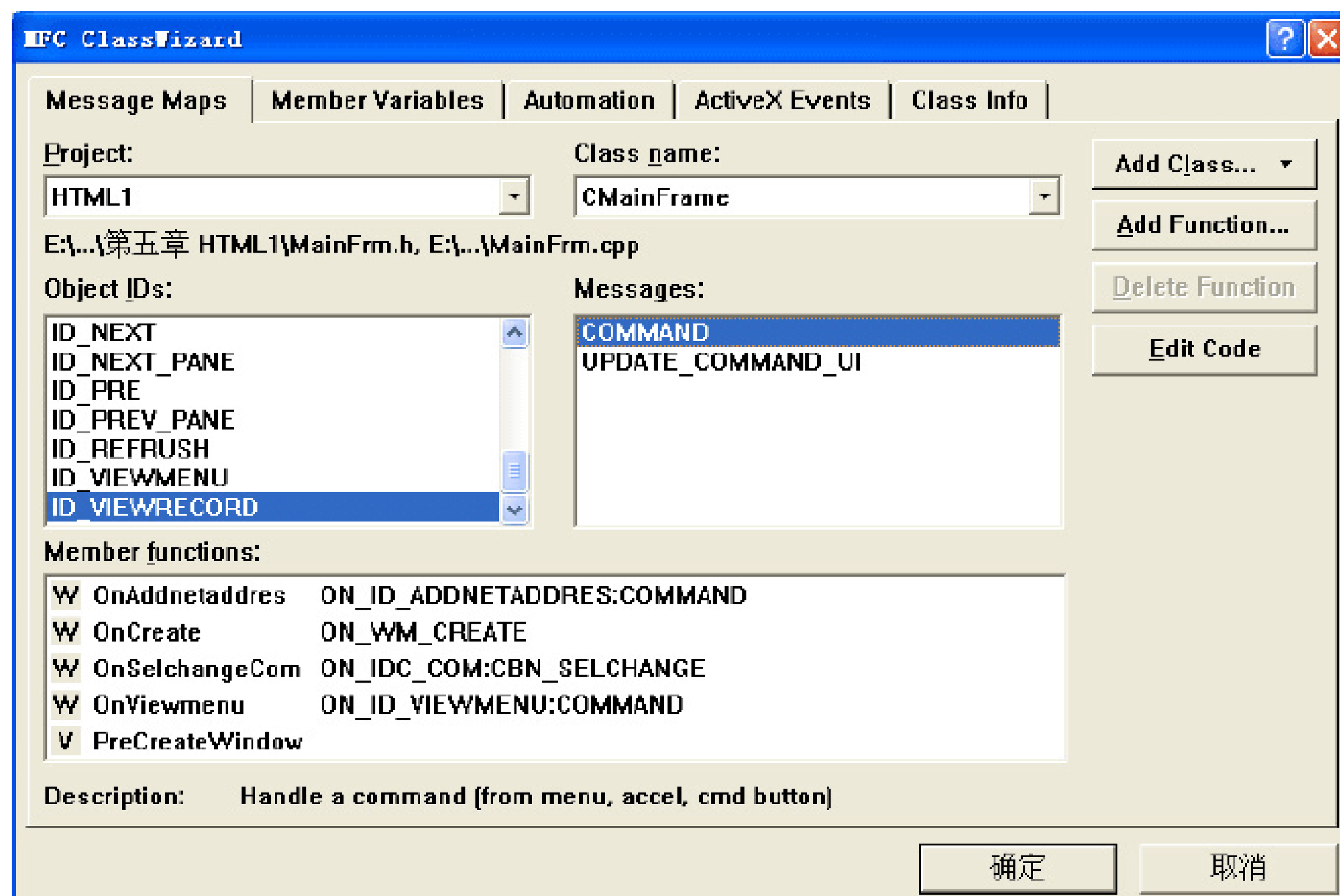


图 5.8　MFC 向导对话框

用户在 Message Maps 选项卡中找到连接按钮的 ID，然后为其添加鼠标单击消息响应函数 OnButton()，然后单击“确定”按钮完成函数添加（其他工具栏按钮添加响应函数的方法与此相同）。函数代码如下：

```
void CMainFrame::OnButton()
{
CString str;                                    //定义字符串变量
GetDlgItem(IDC_COMBO1)->GetWindowText(str);     //获得地址栏输入的字符串
m_view.getpage(str);            //调用视图类自定义函数 getpage()打开网页
...                                             //省略部分代码
}
```

上述代码中，m_view.getpage(str)表示调用视图类中的自定义函数打开指定网页。用户要使用该函数，必须在视图类中进行定义。首先，在 CHTML1View 类的头文件“HTML1View.h”中，定义函数 getpage()，参数类型为 CString 类型。代码如下：

```
class CHTML1View : public CHtmlView     //视图类定义
{
...                                     //省略部分代码
protected:
    CHTML1View();                       //构造函数
public:
    CHTML1Doc* GetDocument();
public:
    void getpage(CString str);          //自定义函数 getpage()
    virtual ~CHTML1View();
...                                     //省略部分代码
};
```

然后在视图类的定义中，实现自定义函数。实现代码如下：

```
void CHTML1View::getpage(CString str)
{
    Navigate2(str,NULL,NULL);           //调用 CHtmlView 类的函数打开网页
}
```

函数 Navigate2()是 CHtmlView 类成员函数，用于打开指定网络地址的网页。其原型如下：

```
void Navigate2( LPCTSTR lpszURL, DWORD dwFlags = 0, LPCTSTR
lpszTargetFrameName = NULL, LPCTSTR lpszHeaders = NULL, LPVOID
lpvPostData = NULL, DWORD dwPostDataLen = 0 );
```

参数 lpszURL 表示网页的网络地址，其他参数均默认为 NULL。在视图类中自定义函数成功之后，将视图类对象设置为框架类 CMainFrame 的成员变量。方法如下：

```
#include <HTML1View.h>                  //包含视图类头文件
class CMainFrame : public CFrameWnd
{
protected:
    CMainFrame();
    DECLARE_DYNCREATE(CMainFrame)
public:
CHTML1View m_view;                      //定义视图类对象
}
```

通过上述代码，用户可以在工程框架类中调用视图类对象的成员函数了。

当用户每次输入网页地址后，程序需要将该地址存入.URL 文件中，以便用户查看浏览记录和向下拉列表框中添加已浏览网页的网络地址。所以需要在连接按钮的响应函数中添加代码，代码如下：

```
void CMainFrame::OnButton()
{
CString str;                                        //定义字符串变量
GetDlgItem(IDC_COMBO1)->GetWindowText(str);    //获得地址栏输入的字符串
m_view.getpage(str);                        //调用自定义函数 getpage 打开网页
CFile file("recode.URL", CFile::modeCreate|CFile::modeReadWrite);
                                            //创建文件，并指定可读可写属性
Str+="\r\n";                                //添加换行符
for(int i=0;i<=str.GetLength();i++)        //循环写入 str 内容
{
char buff=str.GetAt(i);                     //字符指针指向 str
file.Write(&buff,1);                        //将网页地址写入文件中
file.Close();                               //关闭文件
}
```

程序在启动时，还应该从 recode.URL 文件中读取浏览过的网址，并添加到地址栏的下拉列表框中，供用户查看。该功能在函数 CMainFrame::OnCreate()中实现，因为该函数是程序启动后第一个调用的函数。代码如下：

```
int CMainFrame::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
    ...                                                 //省略部分代码
  CFile file("recode.URL", CFile::modeReadWrite);      //创建文件
char buff[100]={0};                                     //定义缓冲区
file.Read(buff,100);                                    //读取文件
while(buff!="\r"&&buff!="\n")                           //判断地址是否结束
{
   GetDlgItem(IDC_COMBO1)->AddString(CString &buff);   //向地址栏添加网址
   file.Read(buff,100);                                 //继续读取文件内容
}
}
```

如果用户从下拉列表框中选择浏览网址，则程序还需要响应组合框的 CBN_SELCHANGE 消息。该响应函数定义如下：

```
void CMainFrame::OnSelchangeCom()
{
CString str;                                        //定义字符串变量
int i=GetDlgItem(IDC_COMBO1)->GetCurSel();         //获得用户单击下拉列表框的索引
    GetDlgItem(IDC_COMBO1)->GetLBText(1,str);      //获取索引处内容
     GetDlgItem(IDC COMBO1)->SetWindowText(str);
     //将获得的网址设置为组合框中的内容
}
```

程序将用户单击处的网址设置为组合框的内容后，单击“连接”按钮，调用 CMainFrame::OnMybutton()函数。现在，程序已经实现了浏览网页和保存查看浏览记录等功能。关于上一步、下一步和刷新等功能的实现非常简单，消息响应函数的创建与连接按钮的响应函数创建方法一样，这里不再赘述。功能代码如下：

```
void CMainFrame::OnNext()                        //下一步按钮消息响应函数
{
m_view. GoForward();                             //调用 CHtmlView 类的成员函数
}
void CMainFrame::OnPre()                         //上一步按钮消息响应函数
{
m_view. GoBack();                                //调用 CHtmlView 类的成员函数
}
void CMainFrame::OnRefrush()                     //刷新按钮消息响应
{
m_view. Refresh();                               //调用 CHtmlView 类的成员函数
}
void CMainFrame::OnViewrecord()                  //浏览记录按钮响应函数
{
CFile file("recode.URL", CFile::modeReadWrite);
                                                 //创建并打开文件
char buff[100]={0};                              //定义缓冲区
file.Read(buff,100);                             //读取文件
char *i=&buff;                                   //声明指针对象指向数据的首地址
CString str;                                     //字符串对象变量
while(i!=NULL)                                   //判断指针是否为空
{
 if(i!="\r"&&i!="\n")                            //判断字符是否为结束符
{
   str+=(CString)i;                              //若不是，则转换为字符串
i+=1;                                            //将指针指向下一个数据
else                                             //若是，则表明已经读出一条网址
{
   file.Read(buff,100);                          //重新读取文件
i=&buff;                                         //重新定义指针
GetDlgItem(IDC_COMBO1)->AddString(str);          //向地址栏添加网址
}
} }}
```

在“浏览记录”按钮的响应函数 OnViewrecord()中，采用了循环读取文件数据，并判断数据是否结束，以便用户判断一条记录的完整性。如果得到一条完整数据，则加入到地址栏中显示，否则将继续循环直到数据结束。到此，工具栏上各个按钮的消息响应函数均以实现，用户可以参照本书所带光盘中的代码进行学习、调试，对读者的学习会有很大的帮助。

5.3.3　如何实现收藏夹的功能

当用户上网时，如果感觉某个网页的内容很吸引人或者具有参考价值，而用户又不愿意记下那些冗长难记的网址。这时，用户就可以使用浏览器上的收藏夹来对该网址进行标记，以方便下次继续浏览该网页。在本节中，将向用户介绍怎样实现收藏夹的功能。

1．添加菜单和按钮

首先，在资源管理器的菜单栏中添加一个菜单项，名称是“网址收藏夹”，属性设置为弹出菜单，其他为默认，如图 5.9 所示。

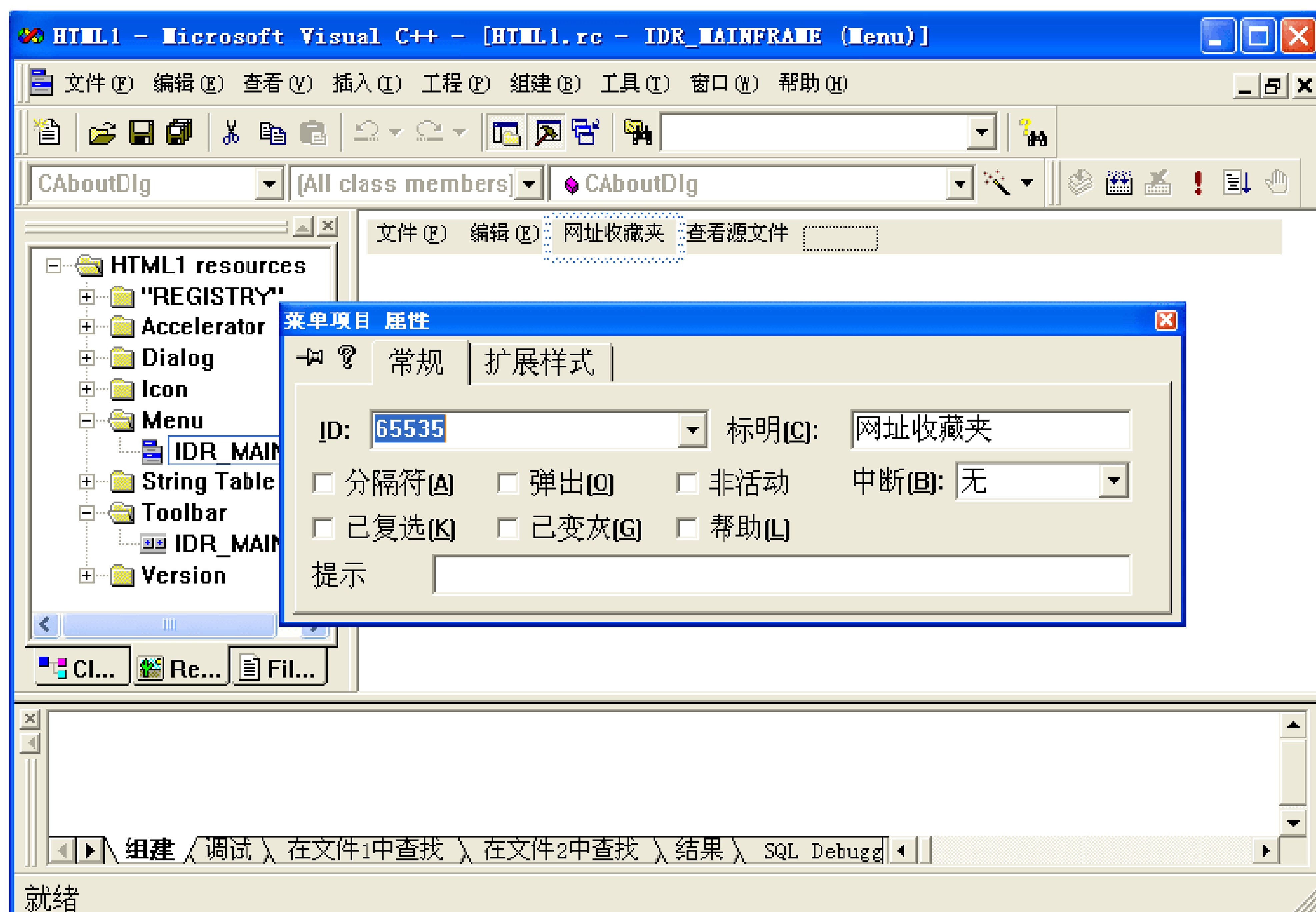


图 5.9　添加弹出菜单

然后，在工具栏中添加一个按钮，名称设置为 T，ID 为 ID_ADDNETADDRES，意思是添加网址到收藏夹，如图 5.10 所示。

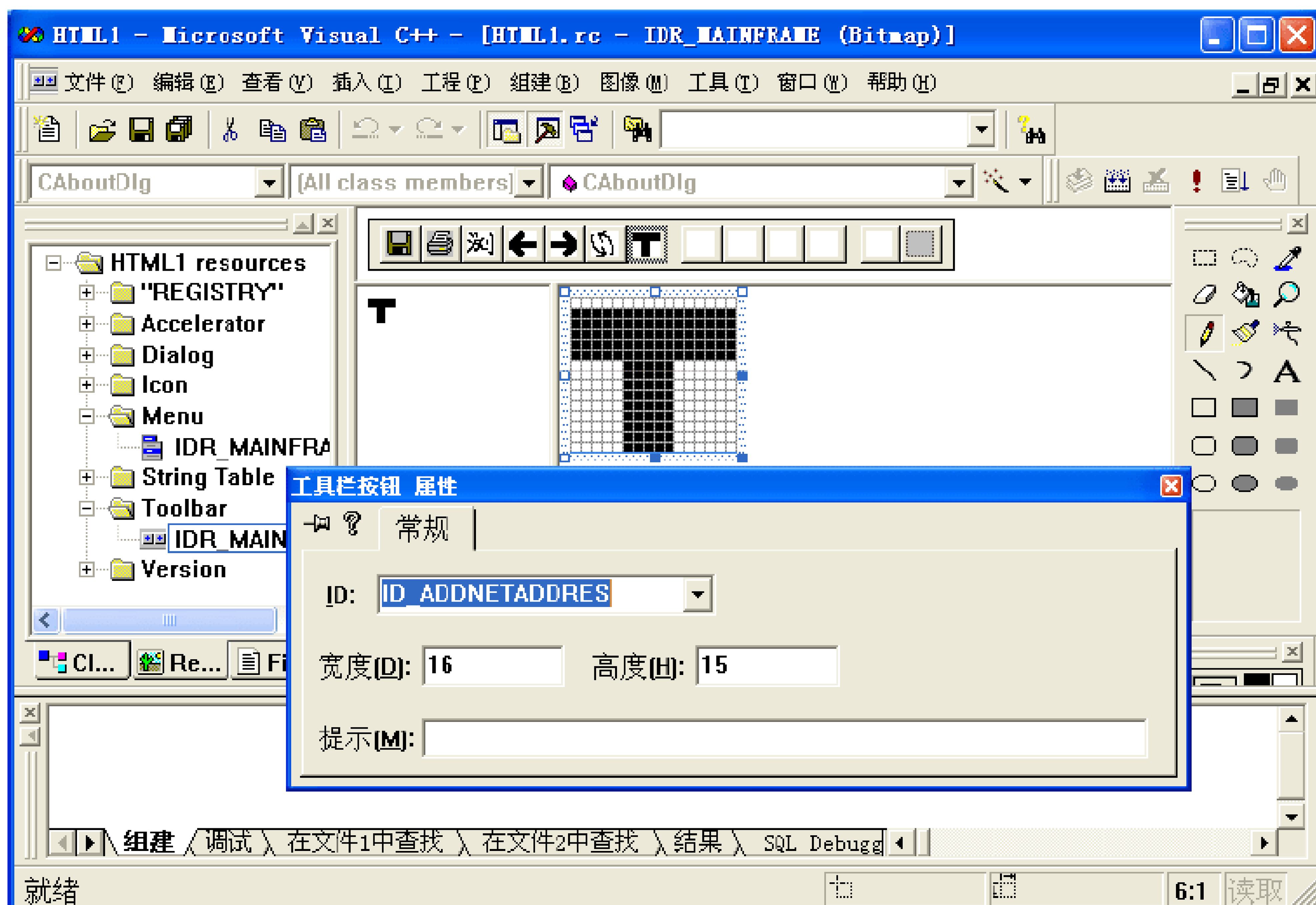


图 5.10　添加工具栏按钮

工具栏中所添加的按钮可以将用户感兴趣的网址添加到“网址收藏夹”菜单下。当用户需要浏览这些网站时，打开“网址收藏夹”菜单选择相应网站即可浏览。

2．添加消息响应函数

在 VC 中，添加工具栏按钮 T 的消息响应函数方法与 5.3.2 节中所讲述的方法一样。如果读者对添加响应函数的方法不清楚，请重新复习 5.3.2 节中的知识。按钮 T 的消息响

应函数定义如下：

```
int i=0;                                                  //定义全局变量 i
void CMainFrame::OnAddnetaddres()
{
 i=::GetMenuItemCount(
::GetSubMenu((HMENU)::GetDlgItem(this->m hWnd,IDR MAINFRAME),3);
                                                          //获取当前菜单总项数
    CString addstr;                                       //定义字符串
this->GetDlgItem(IDC_COMBO1)->GetWindowText(addstr);      //获取地址栏内的网址
     CMenu *menu;                                         //定义菜单指针对象
menu=(CMenu
*)::GetSubMenu((HMENU)::GetDlgItem(this->m_hWnd,IDR_MAINFRAME),3);
                                                          //获取菜单栏的指针
menu->AppendMenu(MF_STRING,i++, addstr);                  //向菜单栏添加网址
}
```

以上代码，通过 CMenu 类的函数 AppendMenu()向收藏夹菜单下添加一个菜单，菜单所显示的文字是用户收藏的网址。

注意：*在程序中使用了符号“::”，表示调用的函数是 Win32 API 全局函数。*

用户将网址添加到收藏夹以后，便可以直接单击菜单中的网址进行浏览。用户单击菜单的消息响应函数是本节的重点。

首先，在 CMainFrame 类的头文件 MainFrm.h 中定义一个弹出菜单的消息响应函数。代码如下：

```
afx_msg void OnMenuClick(int nID);                        //定义响应函数
```

然后，在消息映射里添加菜单命令消息宏 ON_COMMAND_RANGE。代码如下：

```
BEGIN MESSAGE MAP(CMainFrame, CFrameWnd)
//{{AFX_MSG_MAP(CMainFrame)
ON WM CREATE()
ON_COMMAND(ID_NEXT, OnNext)
ON COMMAND(ID PRE, OnPre)
ON_COMMAND(ID_REFRUSH, OnRefrush)
ON_COMMAND(ID_VIEWRECORD, OnViewrecord)
ON BN CLICKED(IDC BUTTON, OnButton)
ON_CBN_SELCHANGE(IDC_COM, OnSelchangeCom)
ON COMMAND(ID ADDNETADDRES, OnAddnetaddres)
ON_COMMAND_RANGE(1,i,OnMenuClick)                     //菜单消息命令宏
//}}AFX MSG MAP
END_MESSAGE_MAP()
```

消息宏 ON_COMMAND_RANGE 的作用是当用户操作一个 ID 范围内的菜单时，调用同一个消息响应函数进行处理。该函数的参数表示当前被用户单击菜单的 ID 号，函数 GetMenuString()通过 ID 号可以获取该菜单上的文字。如果获取成功，则调用 CHtmlView 类的函数 Navigate2()浏览网页。具体代码如下：

```
void CMainFrame::OnMenuClick(int nID)
{
menu.GetMenuString(nID, menustr, MF_BYCOMMAND );          //获取浏览网址
(CHTML1View*) GetActiveView()->Navigate2(menustr,NULL,NULL);
```

```
                                                    //调用函数浏览网页
}
```

上述代码实现了网址收藏夹的功能，用户可以根据代码扩展其功能。例如，将网址写入文件保存在工作目录下，待程序启动时读取文件中数据。这样，用户保存的数据不易丢失，即使程序发生异常，用户也可以从目录下直接打开文件查看。

本节主要讲述了制作网络浏览器界面和各功能函数的具体实现。用户也可以自行添加代码达到自定义效果。

5.4　使用 Microsoft Web 浏览器控件

在 MFC 中，用户可以使用 COM（组件对象）对象来实现网络浏览器。使用 COM 进行编程，不但方便用户缩短开发周期，还可以使用户进一步加深理解面向对象编程的意义。

5.4.1　建立 MFC 工程

在 VC 中，创建网页浏览器的基本步骤与创建 FTP 工程的步骤大体相同，下面将向用户讲解与其不同的创建步骤与设置。

（1）将工程类型设置为单文档，如图 5.11 所示。

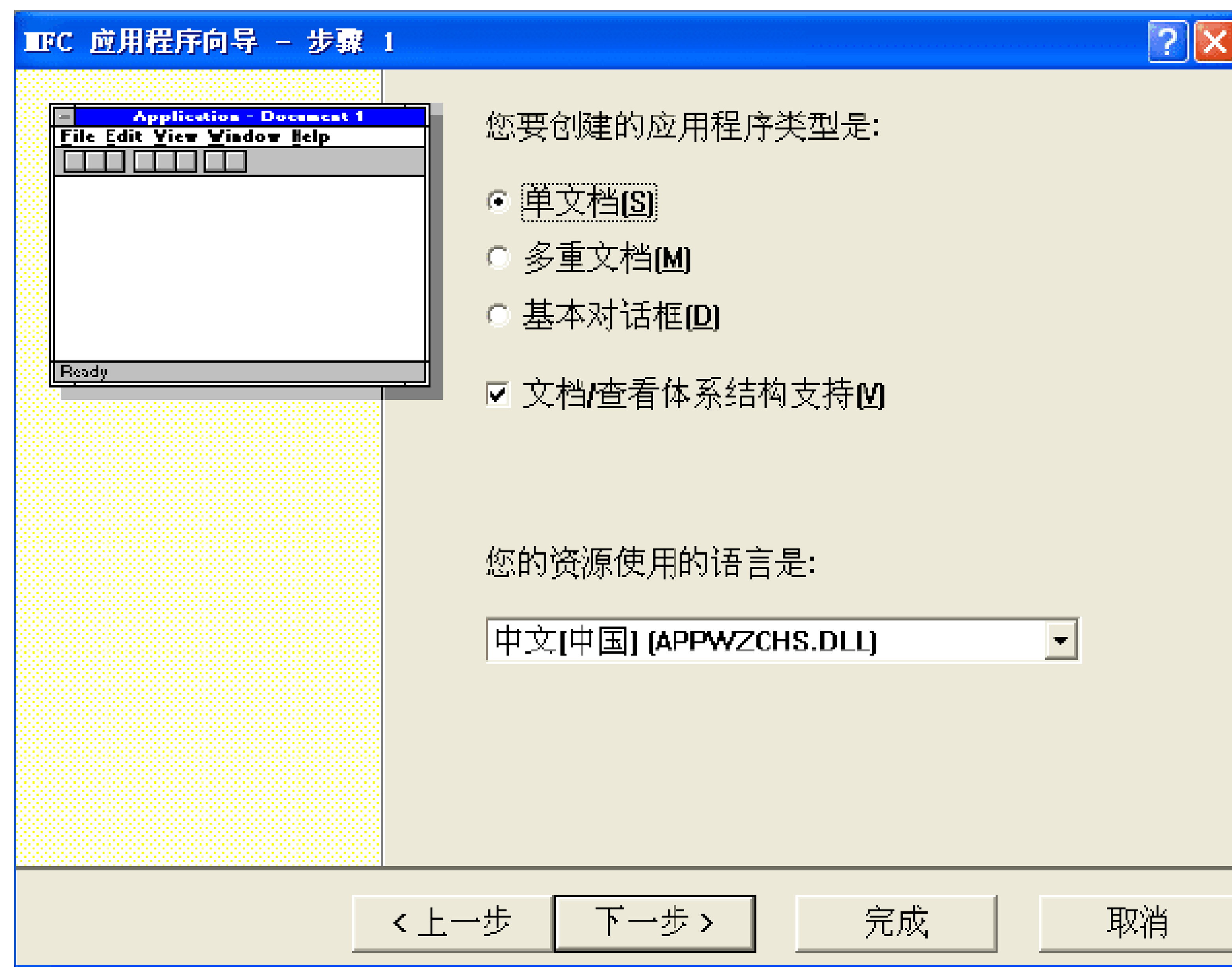


图 5.11　设置工程类型

（2）单击“下一步”按钮到步骤 4，将程序的工具栏样式设置为“类似 IE”，如图 5.12 所示。

（3）工具栏样式设置完成以后，单击“下一步”按钮进入最后一步，如图 5.13 所示。

单击“完成”按钮，完成工程的设置，编译器自动回到 VC 主界面中，并添加“MicroSoft Web 浏览器”组件。



图 5.12　设置工程工具栏样式

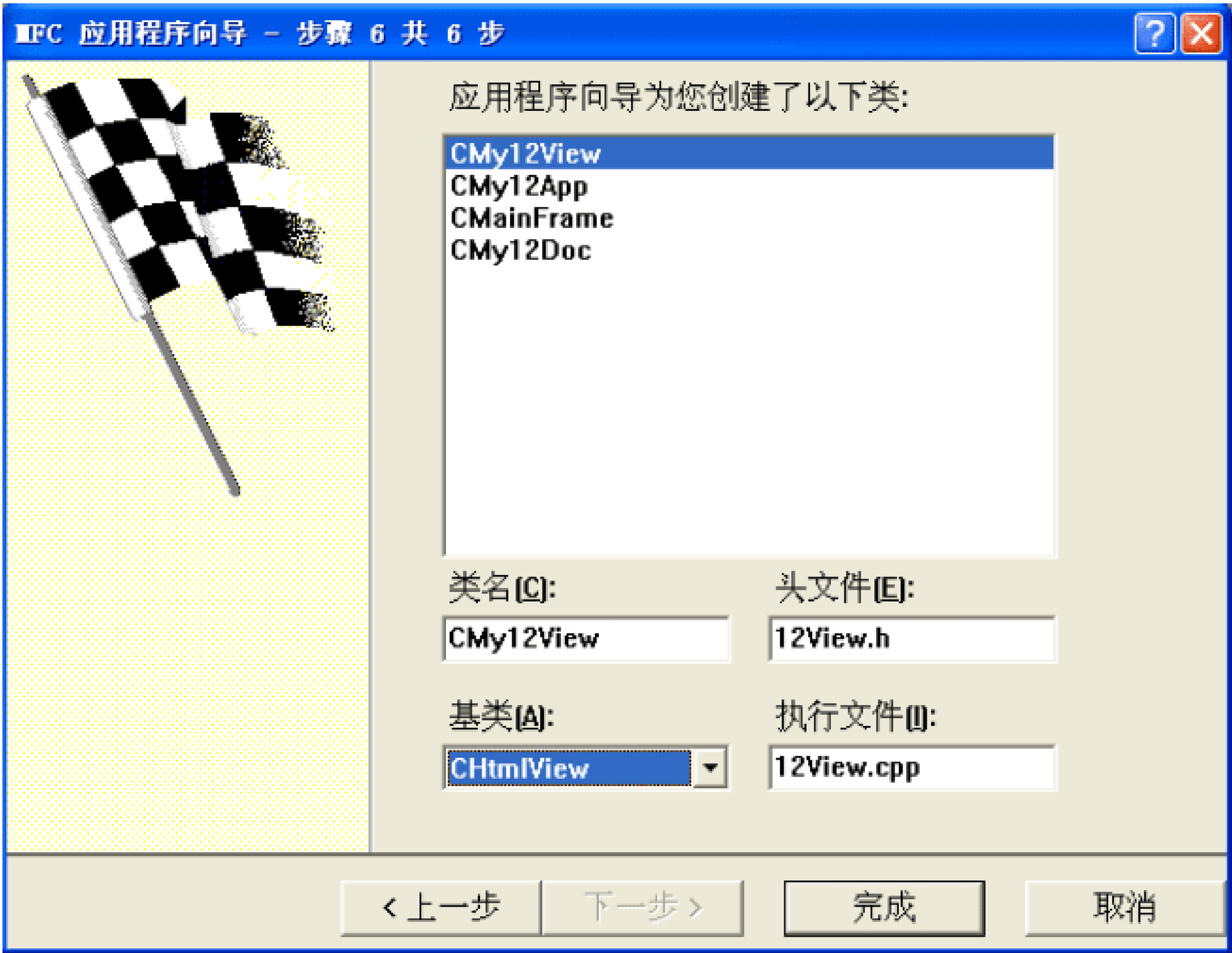


图 5.13　完成设置

5.4.2　添加控件

一般情况下，用户在 VC 中可以利用菜单向工程添加控件。如果该控件没有在程序所运行的系统中进行注册，那么需要用户利用相关工具、代码或者 Windows 命令进行注册控件。控件添加成功，还需要为该控件生成相应的类。具体方法将在本节中讲述。

1．添加COM组件

（1）通过选择“工程”|“增加到工程”命令添加 COM 对象，如图 5.14 所示。

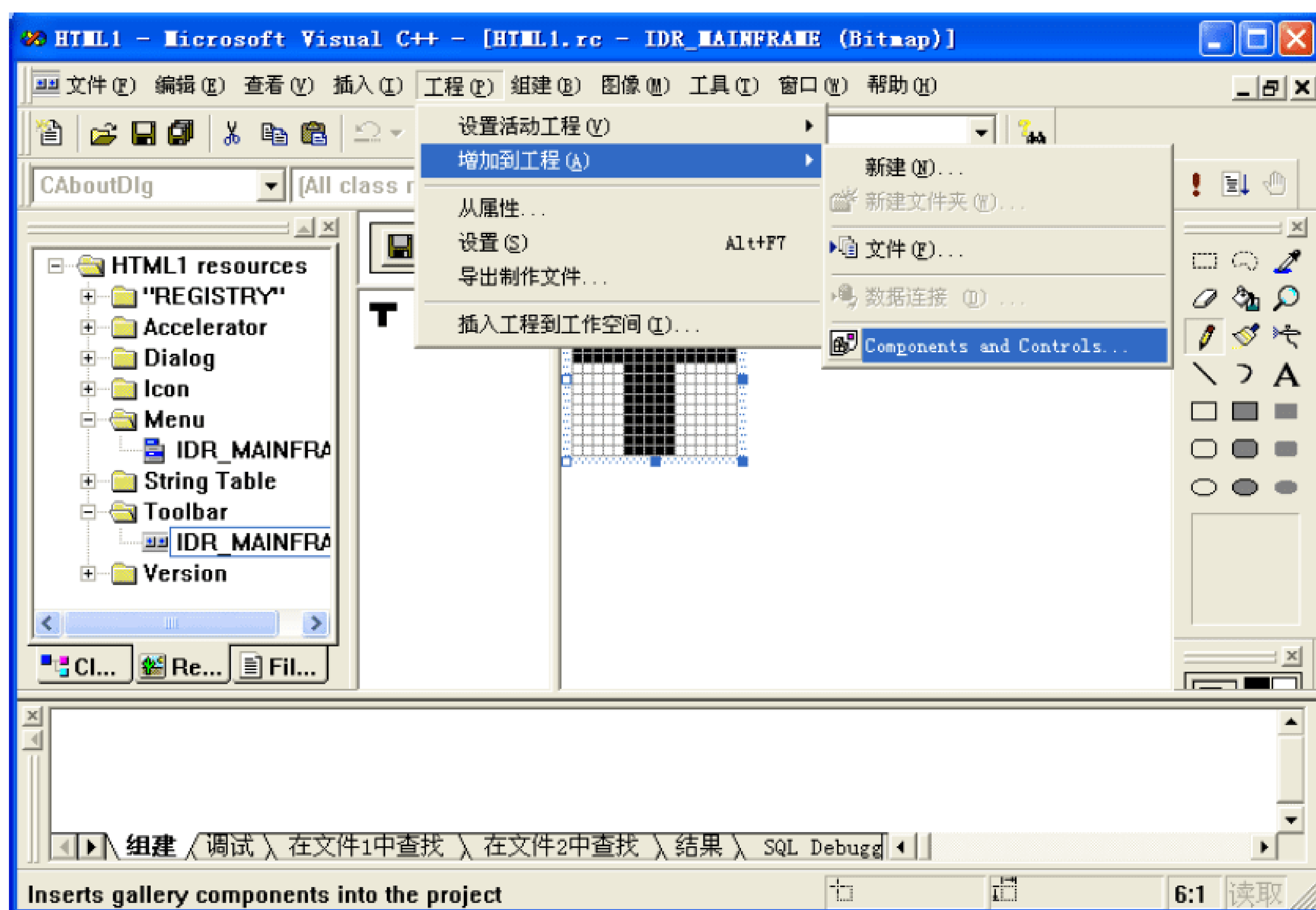


图 5.14　添加 COM 组件对象

（2）选择 Components and Controls Gallery 命令以后，会弹出插入组件对话框，如图 5.15 所示。

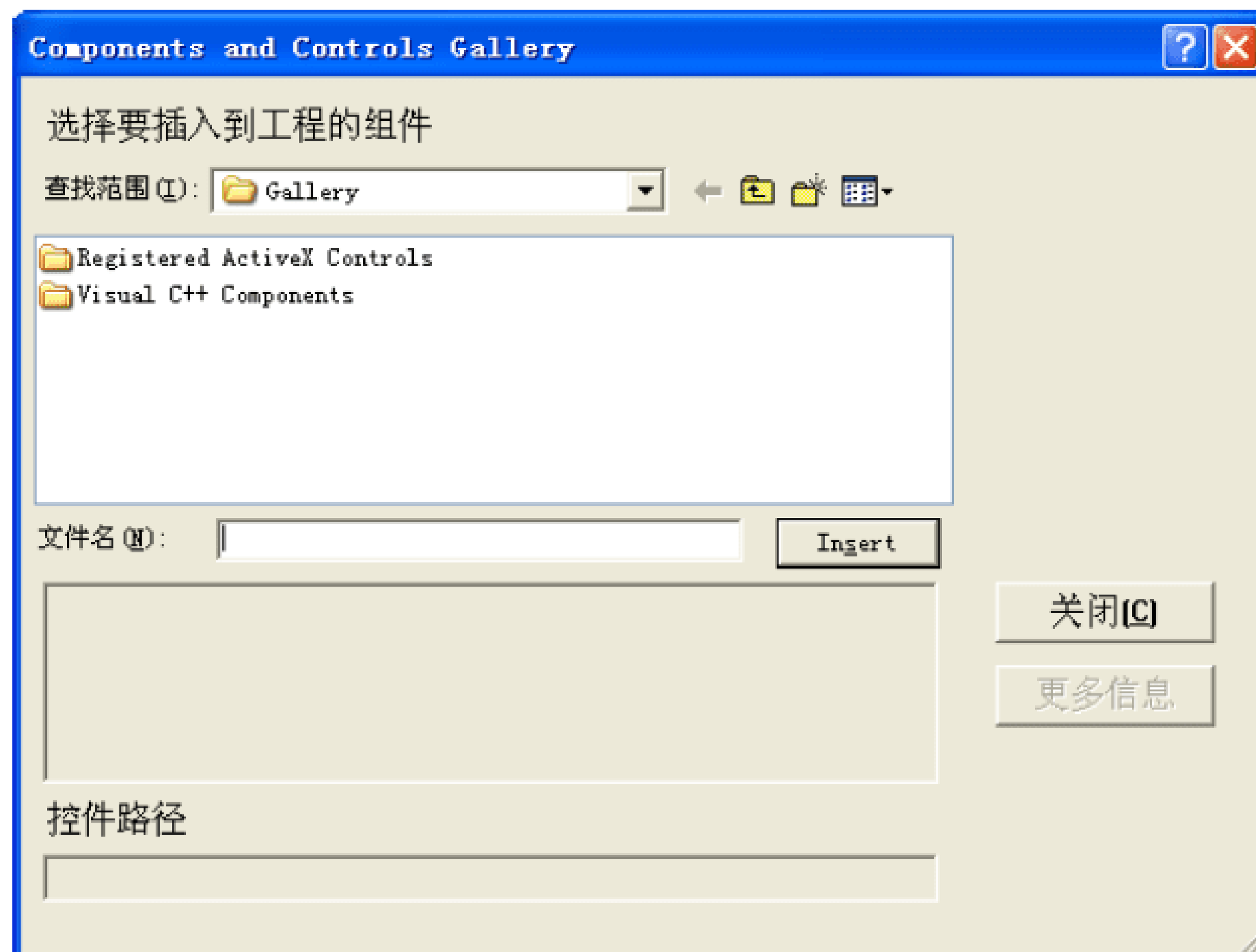


图 5.15　插入组件对话框

（3）双击第一个文件夹，找到 Microsoft Web Browser 组件并单击 Insert 按钮弹出一个询问对话框，直接单击“确定”按钮。这样，用户就可以将 Web 组件插入到工程中，如图 5.16 所示。

（4）将 Web 组件添加到工程中，需要用户为该组件生成一个相应的类。在弹出的配置类对话框中，用户可以修改组件类的名称、头文件名等。在本工程中，使用默认的类名 CWebBrowser2 以及文件名 webbrowser2.h，如图 5.17 所示。

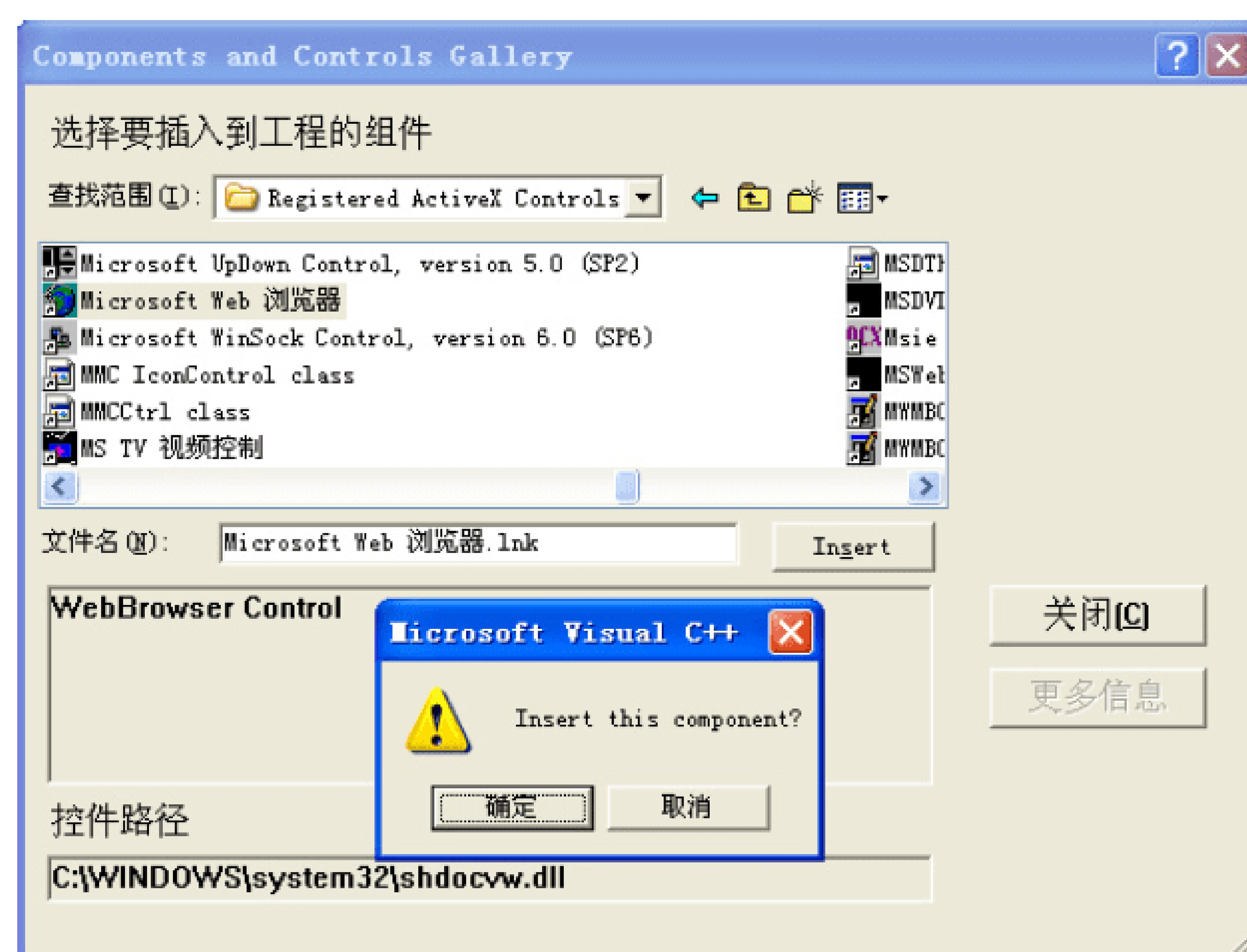


图 5.16　插入 Web 组件到工程

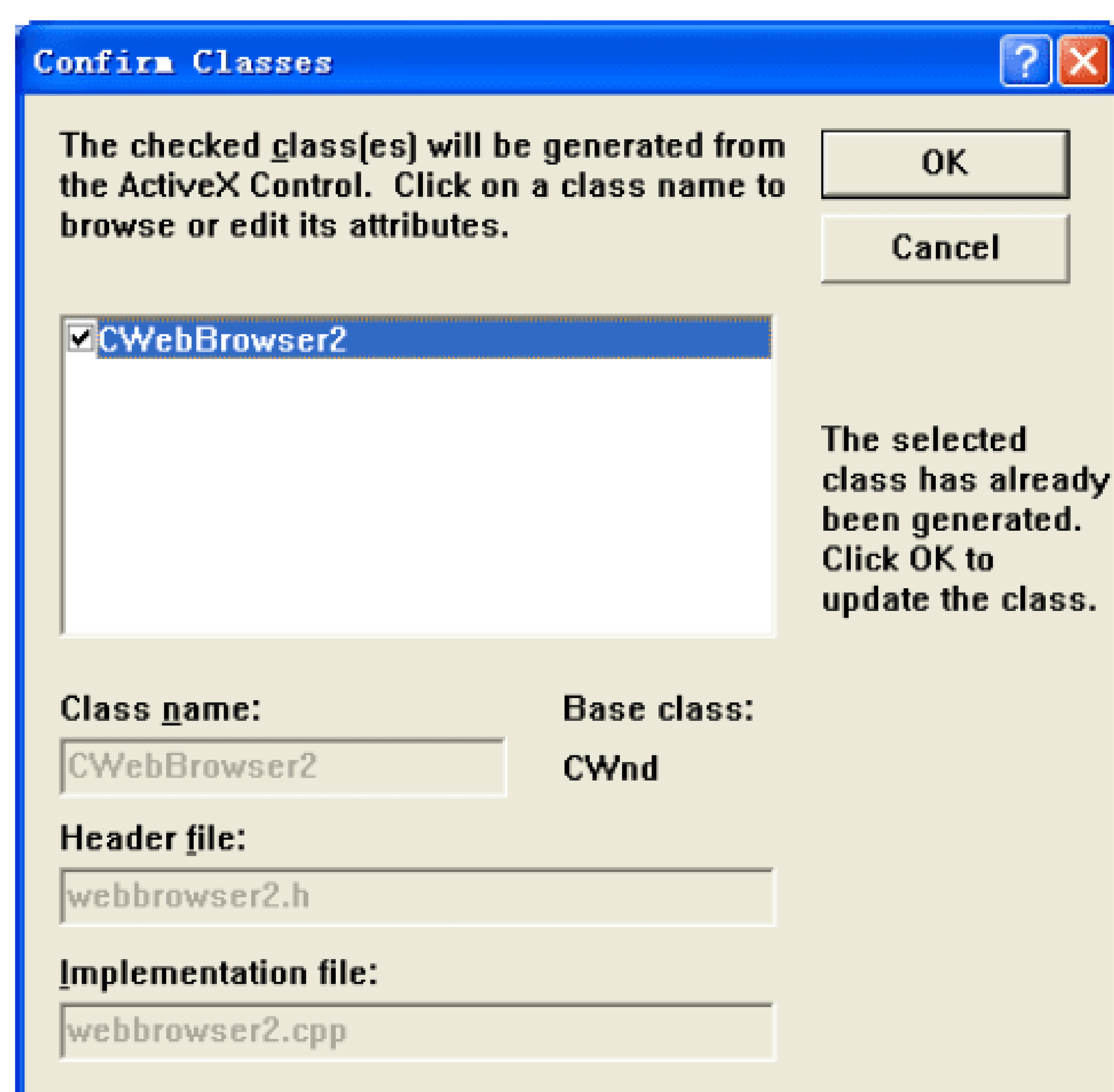


图 5.17　配置类对话框

用户将新建类的信息修改完毕以后，单击 OK 按钮，返回到 VC 主界面，可以在界面左侧的 ClassView 中查看新添加类的声明和定义。

注意：如果用户添加的 COM 组件没有在系统中注册，则需要用户通过相关工具或者代码注册组件，关于此方面的内容请读者参考有关动态链接库的书籍。

2．创建CWebBrowser2对象

如果用户插入 Web 组件成功，那么必须创建组件类对象，再利用该对象调用相应的方法实现网页浏览功能。

（1）在头文件 MainFrm.h 中包含 Web 组件类的头文件 webbrowser2.h。代码如下：

```
#include "webbrowser2.h"                //包含组件类的头文件
class CMainFrame : public CFrameWnd
{
  ...                                   //省略部分代码
}                                       //类定义完成
```

（2）然后，在 CMainFrame 类的定义中声明组件类对象，访问权限设置为 protected。代码如下：

```
class CMainFrame : public CFrameWnd     //定义 CMainFrame 类，继承于 CFrameWnd
{
protected:                              //保护属性
    CWebBrowser2 web;                   //组件类对象
  ...                                   //省略部分代码
}
```

创建 CWebBrowser2 对象时，还可以在该类的构造函数中利用 new 关键字创建其对象。用户将以上步骤操作完成以后，就可以在程序中使用 CWebBrowser2 类方法实现网页浏览功能。

5.4.3　控件对象属性方法

在程序中实现浏览网页等功能是通过 CWebBrowser2 类的方法实现的。在该类中，打开指定网页可以使用函数 CWebBrowser2::Navigate2()。在“连接”按钮的消息响应函数中使用该函数浏览网页，代码如下：

```
void CMainFrame::OnButton()                       //连接按钮响应函数
{
CString str;                                      //定义字符串变量
GetDlgItem(IDC_COMBO1)->GetWindowText(str);       //获得地址栏输入的字符串
web.Navigate2(str,NULL,NULL);             //调用 CWebBrowser2 类函数打开网页
CFile file("recode.URL", CFile::modeCreate|CFile::modeReadWrite);
                                                  //创建文件，并指定可读可写属性
Str+="\r\n";                                      //添加换行符
for(int i=0;i<=str.GetLength();i++)               //循环写入 str 内容
{
char buff=str.GetAt(i);                           //字符指针指向 str
file.Write(&buff,1);                              //将网页地址写入文件中
file.Close();                                     //关闭文件
}
```

在代码中，使用函数 CWebBrowser2 类的 Navigate2()打开指定网页，其参数 str 表示网页的浏览地址，其余参数默认为 NULL。程序将用户输入的网址保存到 recode.URL 文件中供用户查看浏览记录。接下来，将向用户介绍 CWebBrowser2 类的一些常用方法。

```
Void CWebBrowser2:: GoHome();                     //返回主页
Void CWebBrowser2::GoForward();                   //页面前进
Void CWebBrowser2::GoBack();                      //页面返回
Void CWebBrowser2::Refresh();                     //页面刷新
Void CWebBrowser2::Stop();                        //页面停止
```

如果用户想要刷新页面，则在工具栏“刷新”按钮的消息响应函数中，使用

CWebBrowser2 类对象 web 调用 Refresh()函数对页面进行刷新。代码如下：

```
void CMainFrame::OnRefrush()                //刷新按钮消息响应函数
{
web. Refresh();                             //调用 CWebBrowser2 类刷新方法刷新页面
}
```

关于网页浏览器中各个功能的实现与 CWebBrowser2 类的各个函数方法的作用相同，如上面的代码所示。实现浏览器中其他各功能，代码如下：

```
void CMainFrame::OnNext()                   //前进按钮消息响应函数
{
    web.GoForward();                        //调用 CWebBrowser2 类的成员函数
}
void CMainFrame::OnPre()                    //后退按钮消息响应函数
{
    web.Stop();                             //调用 CWebBrowser2 类的成员函数
}
void CMainFrame::OnStop()                   //停止按钮消息响应函数
{
    web.GoBack();                           //调用 CWebBrowser2 类的成员函数
}
void CMainFrame::OnHome()                   //主页按钮消息响应函数
{
    Web.GoHome();                           //调用 CWebBrowser2 类的成员函数
}
```

用户使用 COM 组件对象进行编程，可以实现程序中一些复杂的功能或者界面。在本节中，向用户介绍了在工程中使用 Microsoft Web 浏览器控件访问网页时，需要用到的控件类属性与方法，并且举例说明了一些较常用的 CWebBrowser2 类函数。关于该类的其他功能函数，用户可以参考 MSDN。

5.5　CHtmlView 类

在本章前几节中，已经向用户介绍了实现网页浏览器的几种方法。在 MFC 中，用户还可以使用 CHtmlView 类进行编程。CHtmlView 类是用来显示网页数据的视图类，如果用户在工程中将默认的视图类继承于该类，那么便可以拥有显示网页内容的功能。

5.5.1　CHtmlView 类

CHtmlView 类在 MFC 中是专门用来显示网页的视图类。通常情况下，用户只需将该类作为视图类的父类，便可以调用其类中的函数方法进行网页的显示以及刷新等功能。下面，将向用户介绍该类中部分函数的作用以及使用方法。

如果用户在工程中需要实现连接并打开网页，那么调用该类中的函数 Navigate2()，便可以实现这个功能。其原型如下：

```
void Navigate2( LPCTSTR lpszURL, DWORD dwFlags = 0, LPCTSTR
```

```
lpszTargetFrameName = NULL, LPCTSTR lpszHeaders = NULL, LPVOID
lpvPostData = NULL, DWORD dwPostDataLen = 0 );
```

该函数的作用是连接并打开指定网页。其中，参数 lpszURL 为将要打开的网页地址，其他参数均为 NULL。例如，用户需要打开地址为“www.163.com”的网页，则将参数 lpszURL 设置为该地址即可。代码如下：

```
CHTML1View:: Navigate2("www.163.com",0,NULL, NULL, NULL,0);
```

上面的代码运行之后，将在工程视图中打开并显示地址为“www.163.com”的网页内容。当用户浏览网页时，可以使用该类中提供的刷新功能获取更新后的当前网页内容，也可以在工程中查看已经浏览过的网页等。代码如下：

```
void CHTML1View::OnRefresh()                    //刷新网页内容
{
    this->Refresh();
}
void CHTML1View::OnPre()                        //查看上一步浏览的网页
{
this->GoBack();
}
void CHTML1View::OnNext()                       //查看下一步
{
    this->GoForward();
}
```

在本章的实例程序中，主要使用以上函数实现网页的浏览等功能。如果用户需要了解关于该类的其他知识，请查阅 MSDN 或参考书籍。本章不再对该类进行详细讲解。

5.5.2　建立继承关系

一般情况下，用户在生成工程项目时，通过应用程序向导将程序视图类的父类设置为 CHtmlView 类，其余步骤均与 5.3 节中相同，如图 5.18 所示。

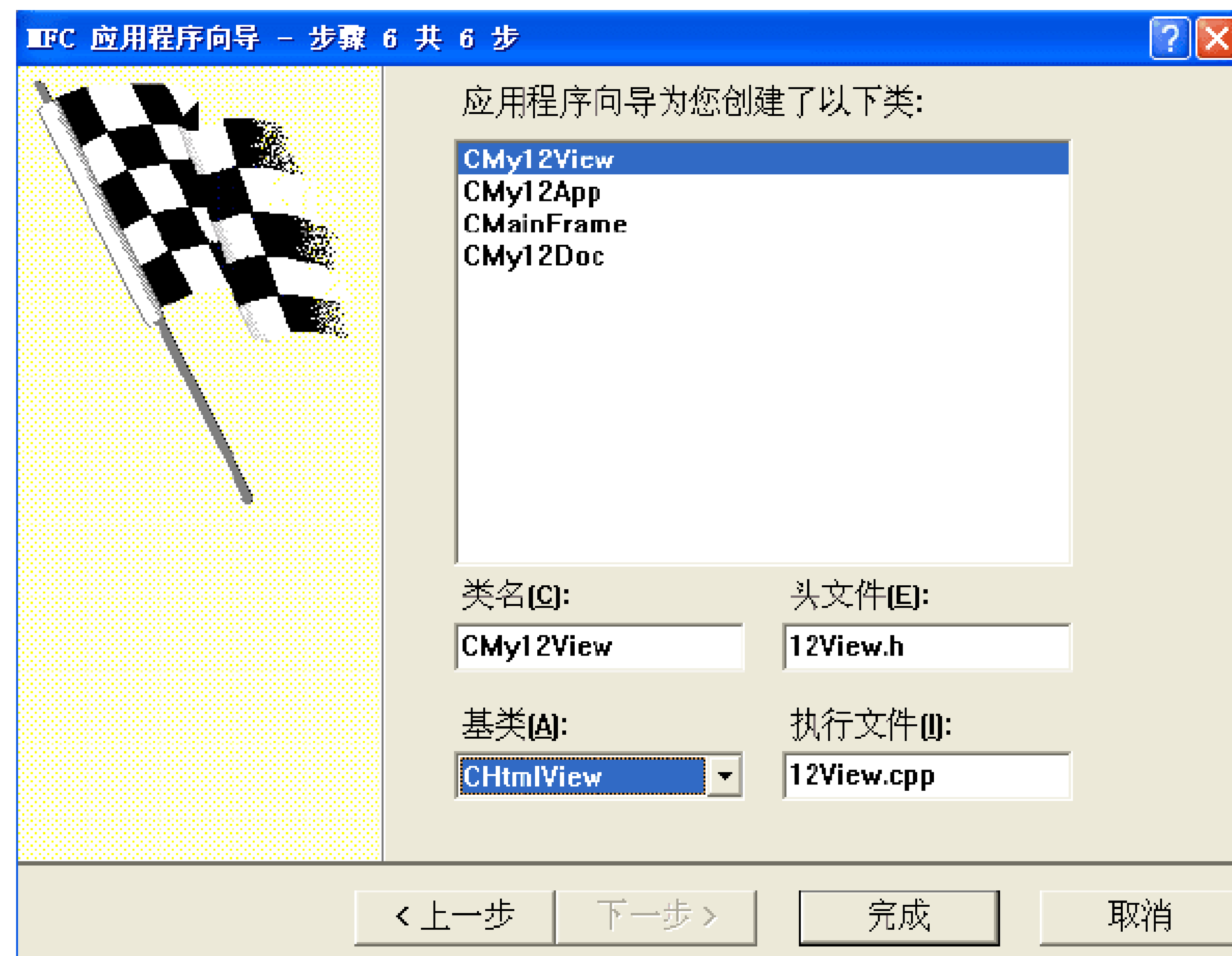


图 5.18　设置工程视图基类

用户除了通过应用程序向导修改基类外，也可以通过手工修改代码完成。首先，在头文件 HTML1View.h 中手工将 CHTML1View 类的父类修改为 CHtmlView。代码如下：

```
class CHTML1View : public CHtmlView     //将视图类的父类修改为 CHtmlView
{
...                                     //省略部分代码
}
```

然后，在文件 HTML1View.cpp 中将动态创建宏和消息映射宏中的父类名修改为 CHtmlView 类。代码如下：

```
IMPLEMENT_DYNCREATE(CHTML1View, CHtmlView)          //动态创建宏
BEGIN_MESSAGE_MAP(CHTML1View, CHtmlView)            //消息映射宏
    //{{AFX_MSG_MAP(CHTML1View)
    //}}AFX_MSG_MAP
    // Standard printing commands
    ON_COMMAND(ID_FILE_PRINT, CHtmlView::OnFilePrint)
END_MESSAGE_MAP()
```

通过以上两种方法可以修改工程中视图类和 CHtmlView 类的继承关系，并且视图类继承了 CHtmlView 类中的一些功能函数。

5.5.3　地址栏消息响应

在工程界面中，对于地址栏的消息响应是浏览器中非常重要的。用户在地址栏中输入网址，单击“连接”按钮后，程序打开指定网页内容。CHtmlView 类中的函数 Navigate2()可以显示网页。代码如下：

```
void CHTML1View::OnButton()
{
CString str;                                         //定义字符串变量
GetDlgItem(IDC_COMBO1)->GetWindowText(str);      //获得地址栏输入的字符串
this-> Navigate2("www.163.com",0,NULL, NULL, NULL,0);
                                    //调用视图类函数 Navigate2()显示网页
...                                                  //省略部分代码
}
```

在代码中，程序利用 this 指针调用函数 Navigate2()显示网页内容。关于其他消息响应函数与 5.2 节中相同，所以本节不再进行讲解。请读者复习前面的相关内容。

5.5.4　实现查看源文件功能

在本小节中，需要向用户讲解在网页浏览器中如何实现查看网页源文件的功能。当用户需要查看某个网页的源代码时，该功能会调用系统中的记事本程序打开该网页的源代码，如图 5.19 所示。

用户要实现如图 5.19 中的效果，需要在菜单栏中添加一个“查看源文件”菜单。具体步骤如下：

（1）在 VC 资源管理器中，插入一个菜单选项，名称为“查看源文件”，ID 设置为 ID_VIEWMENU，如图 5.20 所示。

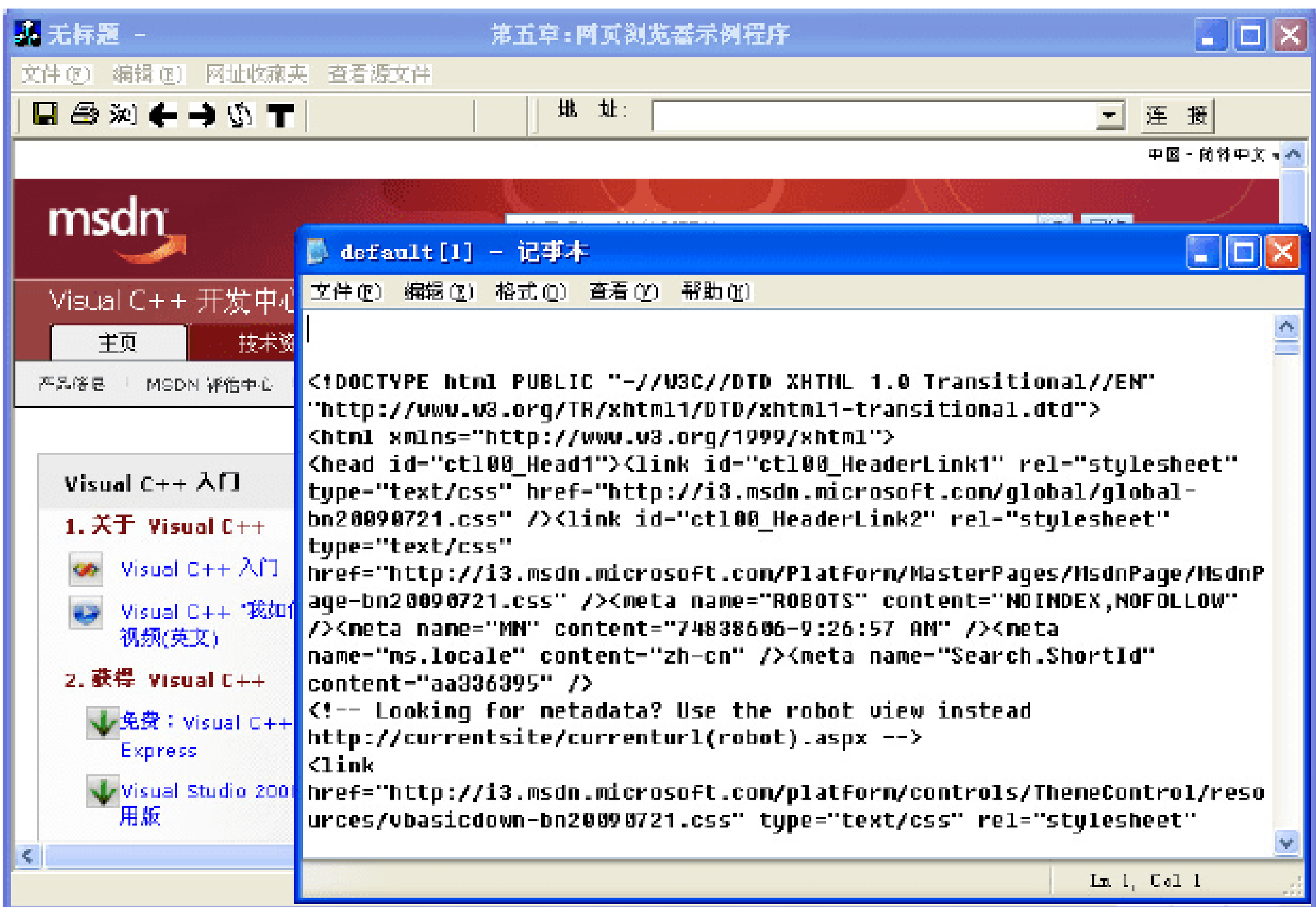


图 5.19　查看网页源文件

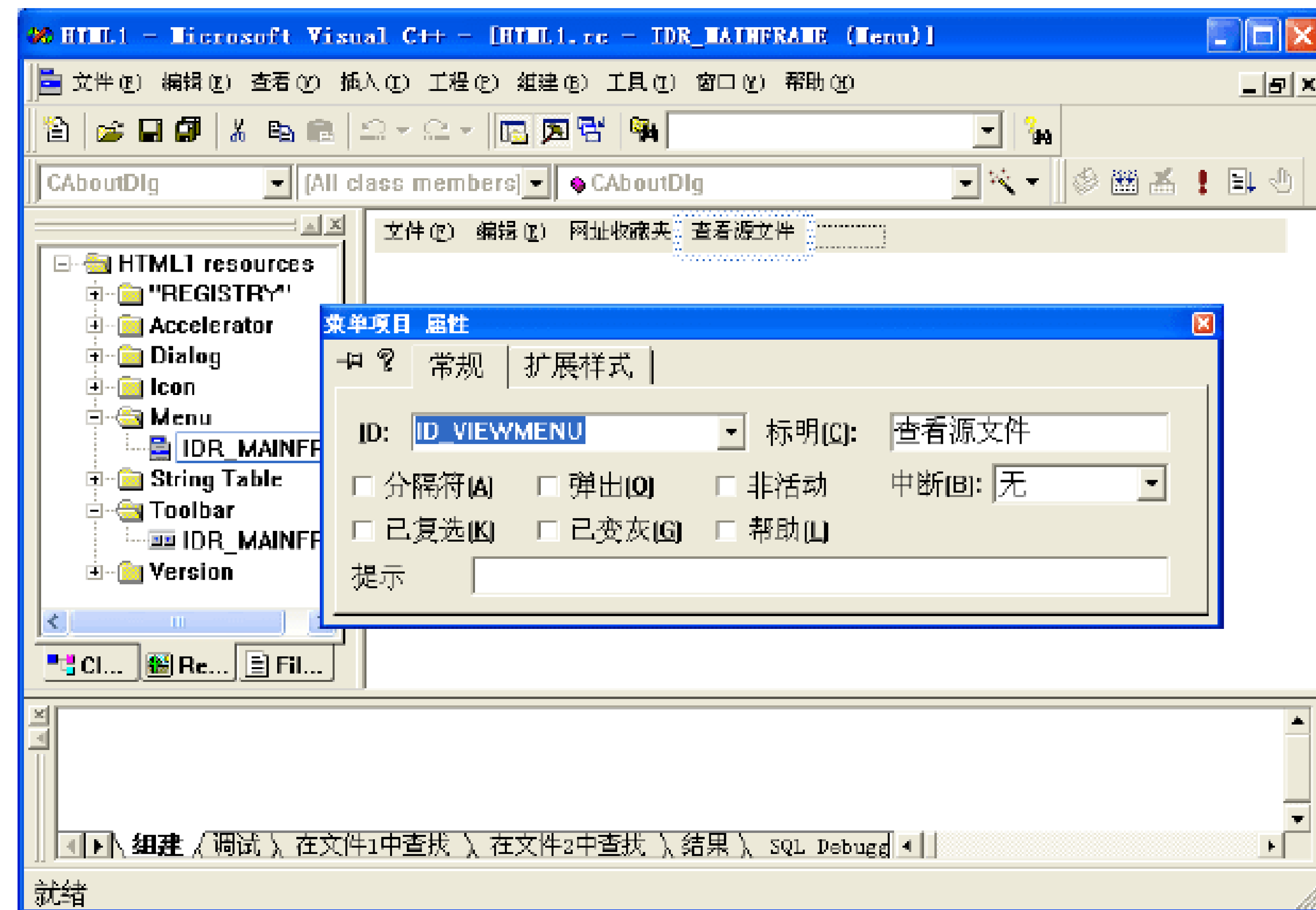


图 5.20　插入菜单

（2）添加消息映射函数，如图 5.21 所示。

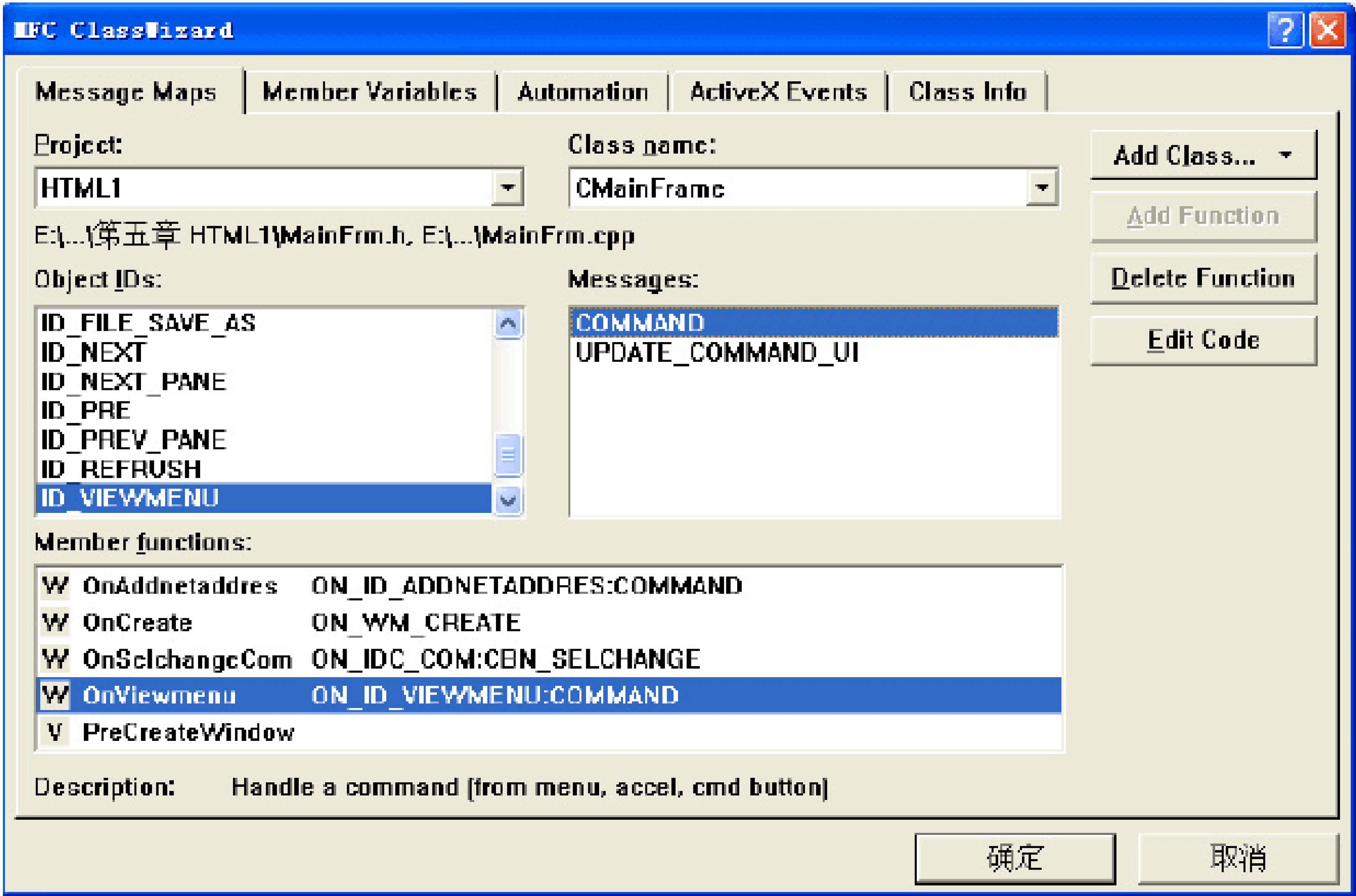


图 5.21　添加消息映射

（3）单击“Edit Code”按钮，可以定位到函数定义处添加自定义代码。在本函数中，用户需要调用系统的记事本来打开网页的源代码。

在 Windows 系统中，用户通常可以使用 Shell 函数调用外部.exe 程序。在这里，向用户介绍 Win32 API 函数中的 ShellExecute 函数，其原型如下：

```
HINSTANCE ShellExecute(
    HWND hwnd,                      //父窗口的句柄
    LPCTSTR lpOperation,            //表示将操作的方式
    LPCTSTR lpFile,                 //表示将操作的文件名
    LPCTSTR lpParameters,           //表示传入程序的文件名
    LPCTSTR lpDirectory,            //所调用程序所在的目录名
    int nShowCmd                    //以何种方式显示程序
);
```

该函数调用成功，则返回被调用程序的应用程序句柄。函数部分参数如下：

- lpOperation 表示将以何种方式操作.exe 程序。常用取值如表 5.5 所示。

表 5.5　参数lpOperation常用取值

取　　值	意　　义	取　　值	意　　义
open	以打开方式	print	以打印方式

- lpParameters 表示传入被启动程序执行的文件名，可以取值为 NULL。
- nShowCmd 表示以何种方式显示程序，该参数取值或组合值，如表 5.6 所示。

表 5.6　参数nShowCmd常用取值

取　　值	意　　义	取　　值	意　　义
SW_HIDE	以隐藏方式打开程序	SW_SHOW	以显示方式打开程序
SW_MAXIMIZE	以最大尺寸打开程序	SW_SHOWDEFAULT	以默认方式打开程序
SW_MINIMIZE	以最小尺寸打开程序		

使用该函数进行调用系统中的记事本程序。代码如下：

```
::ShellExecute(this->m_hWnd,"open","notepad.exe",NULL,NULL,SW_SHOW);
//调用函数启动记事本程序
```

上面代码的作用是以打开模式显式地启动记事本程序。如果需要启动的记事本打开用户指定的文件，则需要将参数 lpParameters 设置为需要启动的文件名。代码如下：

```
::ShellExecute(this->m_hWnd,"open","notepad.exe","d:\我的文档\桌面\新建文本
文档.txt",NULL,SW_SHOW);                  //调用函数启动记事本程序打开文件
```

执行上面的代码，结果是启动记事本程序同时在记事本中打开指定位置的文件。在本章程序中利用该函数实现查看源代码功能需要将接收到的服务器响应数据保存在文件中。保存文件的部分代码如下：

```
...                                       //省略部分代码
char sch[2048]={0};
::recv(s,&sch,2048, MSG_PEEK);            //接收服务器返回的响应数据
CFile file("数据文件.txt", CFile::modeCreate|CFile::modeReadWrite);
                                          //创建文件保存数据
```

```
file.Write(&sch,2048);                      //将数据写入文件
file.Close();                               //关闭文件
```

代码中，函数 recv()的作用是在已经连接的套接字上接收数据，参数 s 表示套接字句柄。将上述保存的数据文件使用记事本程序打开，代码如下：

```
::ShellExecute(this->m_hWnd,"open","notepad.exe",
"数据文件.txt",NULL,SW_SHOW);                 //启动记事本程序打开文件
```

在菜单“查看源文件”的消息响应函数中实现完整的查看源文件功能。代码如下：

```
void CMainFrame::OnViewmenu()               //查看源代码函数
{
    char sch[2048]={0};
    CFile file("原始数据文件.txt", CFile::modeCreate|CFile::modeReadWrite);
    file.Read(&sch,2048);
CString *str=(CString)&sch;
if(str+8==200)                              //消息响应码位于第八位
{
   while(str!=EOF)
{
    if(str!="<"&& str!="h")                 //判断字符数据
      {
CFile file("数据文件.txt", CFile::modeCreate|CFile::modeReadWrite);
                                            //创建文件保存数据
file.Write(str,sizeof(str));                //将数据写入文件
        str+=2;                             //移动数据指针
}}
file.Close();                               //关闭文件
::ShellExecute(this->m_hWnd,"open","notepad.exe","数据文
件.txt",NULL,SW_SHOW);
                                            //启动记事本程序打开文件
}
}
```

首先，程序创建原始数据文件，然后再从该文件中读取服务器返回的有效数据并存入数据文件中。数据读取完毕以后，应该关闭文件，调用 API 函数 ShellExecute()启动记事本程序打开数据文件并显示其中的内容。

5.5.5　实现刷新功能

在 CHtmlView 类中，用户可以调用 Refresh()函数实现刷新功能。实际上，网页刷新显示的原理是客户端重新向服务器发出数据请求，待服务器返回数据后，客户端更新界面以便显示数据。根据这个原理，在本工程中，实现自定义的 MyRefresh()函数，代码如下：

```
void CMainFrame::MyRefresh()                     //自定义函数
{
   CString str;                                  //定义字符串变量
GetDlgItem(IDC_COMBO1)->GetWindowText(str);      //获得地址栏输入的字符串
GetActiveView()->GetDocument()->UpdateAllViews();
                                                 //调用函数更新界面，显示数据
this->Navigate2(str,NULL,NULL);          //调用视图类函数 Navigate2()显示网页
}
```

代码中，首先获取当前地址栏中的网址，再使用视图类函数 GetDocument()获得文档类对象的指针。利用该指针调用 UpDateAllViews()函数更新客户端界面，显示数据。

如果用户调用 CHtmlView 类中的函数 Refresh()进行数据刷新，其原理与自定义函数 MyRefresh()基本一致。代码如下：

```
void CMainFrame::Refresh()
{
GetActiveView()->Refresh();                //调用 Refresh()函数进行刷新
}
```

MyRefresh()和 Refresh()函数的差别在于两者的工作机制不同。前者首先对界面进行更新，然后再调用 CHtmlView 类的 Navigate2()函数重新打开网页。而后者是直接调用 CHtmlView 类的刷新函数 Refresh()。因此，读者可以根据自定义的刷新函数 MyRefresh()，学习有关网页刷新的基本原理以及编程方法等。

5.6　小　　结

在本章中，向用户讲解了网页浏览器的工作原理，根据其原理使用 POST 和 GET 模式向服务器传送数据。通过 HTML 代码向用户介绍了服务器接受请求以后返回的响应数据结构。在 VC 中，从用户创建工程界面编程和每一个消息响应函数的实现上，都向用户进行了非常细致的讲解。用户通过本章的学习，应该了解 HTTP 的消息请求以及响应数据格式等基本结构，并且掌握基本的 HTTP 编程原理和扩充这一知识的能力。

同时，本章中的实例程序也可以供用户做实际项目时进行参考或者扩充。在随书光盘中，实例可以直接运行或编译，但是由于实例程序是在 VC 下编写的，所以用户在 VC 8.0 中编译时需要进行项目格式的转换。关于转换方式请读者参考相关书籍。

第 6 章　网络通信器

现在，有许多即时通信软件在大家的生活中非常常见，并且起着很大的作用。即时通信软件可以让用户之间快速地进行交流沟通，也正是因为这个原因使人们对即时通信软件的需求非常大，对其功能要求也很苛刻。在本章中，将向用户介绍实现即时通信功能的软件编程方法以及通信原理。

6.1　通 信 原 理

网络通信软件的数据通信是通过网络套接字进行的。根据该原理，其编程步骤应分为创建套接字、在套接字上进行收发数据、关闭套接字等操作。在这里需要用户注意：如果在服务器端进行编程，成功创建套接字以后，需要将本地地址与端口号绑定到已经创建的套接字。

在 VC 中，创建基于对话框模式的应用程序，利用资源管理器对程序界面进行整理，使界面整齐、美观。但是，限于笔者的美工水平，所设计出来的程序界面仅供用户学习和参考，笔者主要讲述程序设计方法等。如果用户对界面不够满意，可以对随书光盘中的本实例界面重新进行设计。客户端和服务器端界面效果分别如图 6.1 和图 6.2 所示。

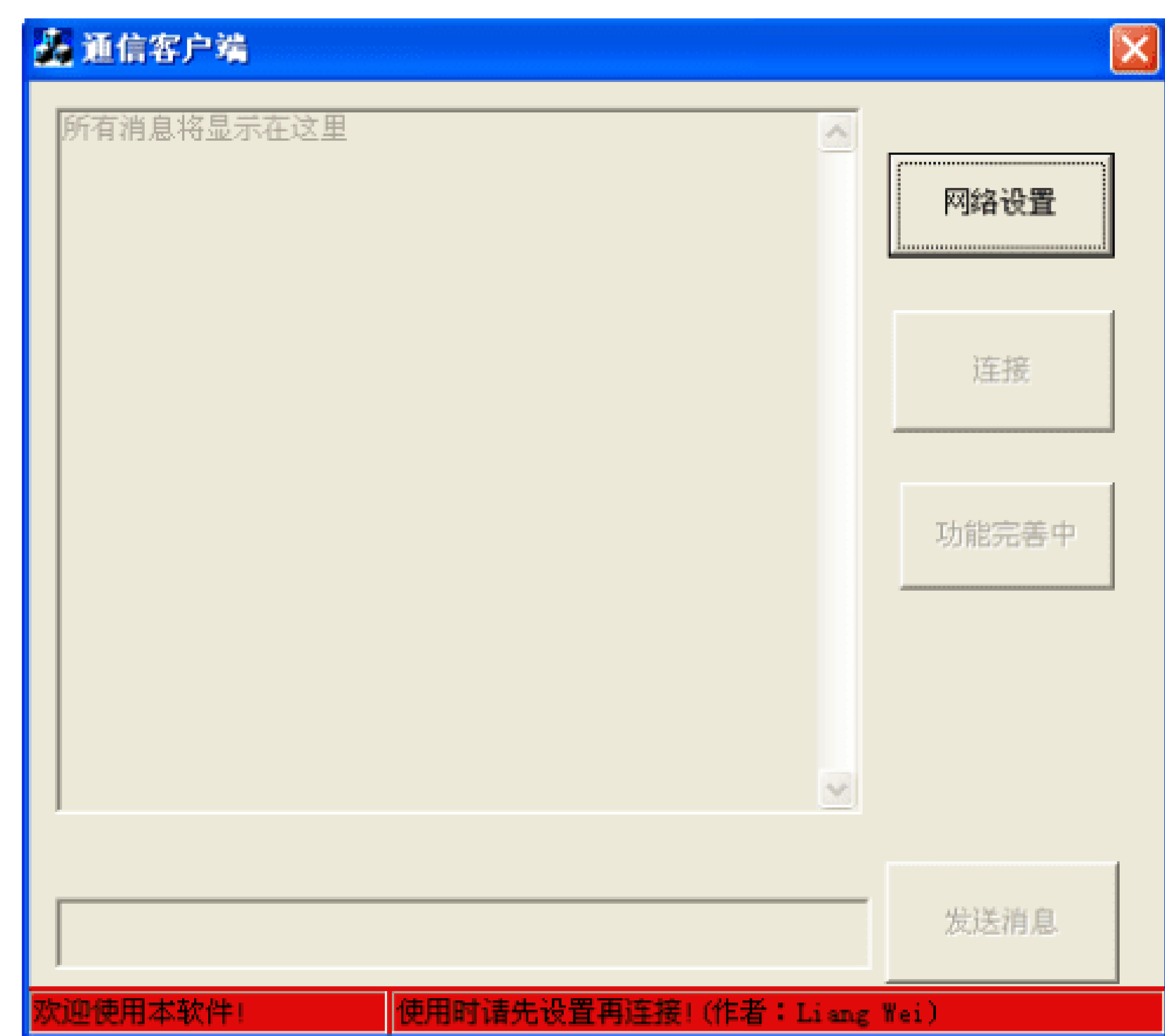


图 6.1　客户端界面

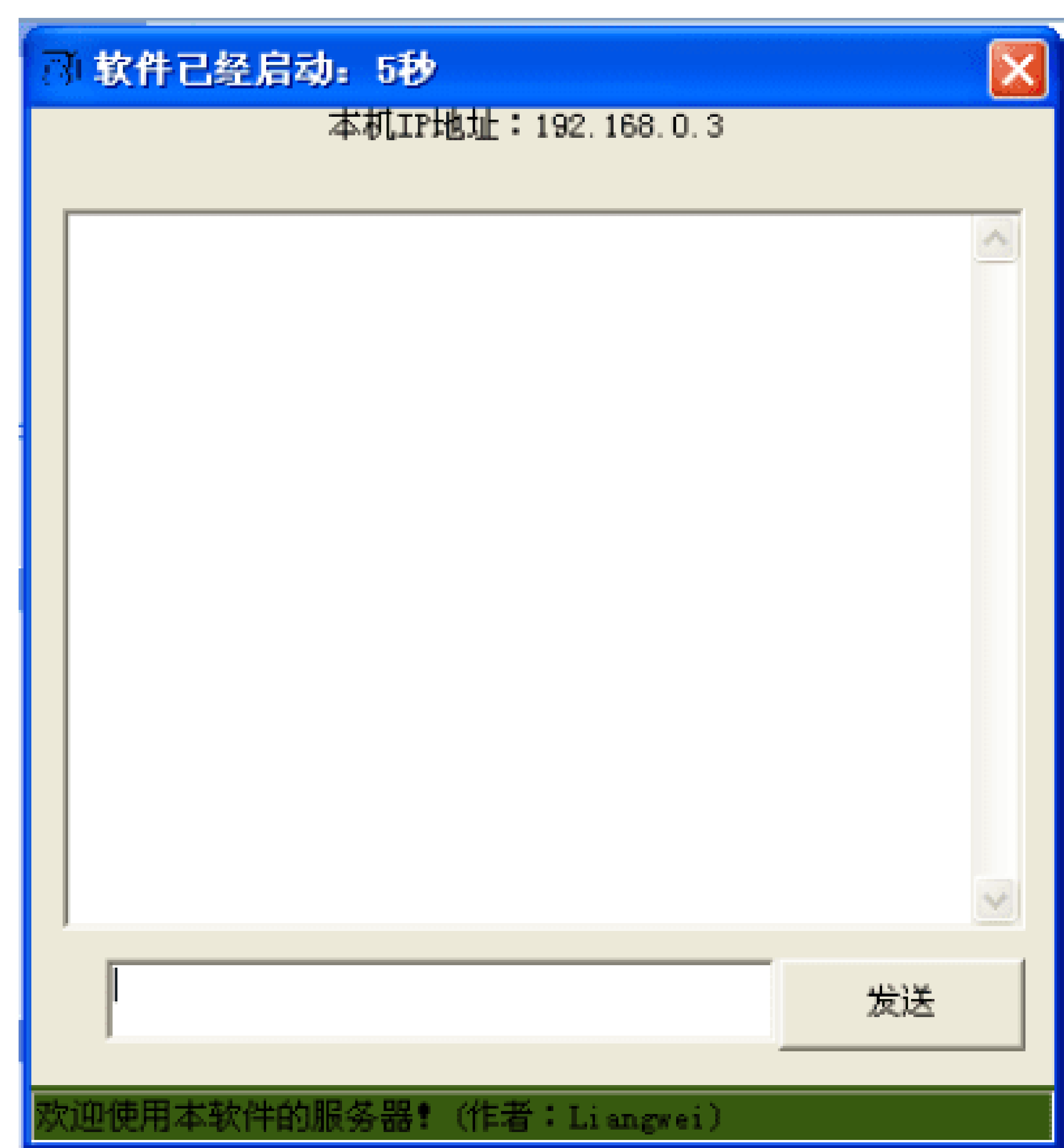


图 6.2　服务器端界面

注意：用户在实际使用时，应该首先启动服务器，然后再启动客户端。否则，客户端将不能连接服务器。

6.1.1　通信连接

在通信软件初始化时，客户端连接服务器的过程是该应用程序初始化的第一步，也是很重要的一步。用户利用 API 函数创建套接字，需要对套接字库进行初始化。代码如下：

```
...                             //省略部分代码
WSADATA data;
DWORD ss=MAKEWORD(2,0);         //指定套接字库版本号
  ::WSAStartup(ss,&data);       //初始化套接字库
```

当程序正常退出或者遇到其他情况退出时，用户应该对已经初始化的套接字库进行释放。示例代码如下：

```
...                             //省略部分代码
WSACleanup();                   //释放套接字库
```

1. 创建套接字

用户对套接字库初始化成功后，便可以调用前面所介绍的函数创建套接字了。对于服务器和客户端而言，服务器的套接字分为连接套接字和数据收发套接字。因为作为服务器不可能只响应一个客户端的连接请求，所以创建连接套接字对所有的连接请求进行响应。下面，将分别向用户介绍创建客户端和服务器端套接字的具体方法。

（1）创建客户端套接字

对于创建客户端套接，需要用户指定服务器的 IP 地址和数据通信端口号。代码如下：

```
...                                                 //省略部分代码
SOCKET s;                                           //声明套接字对象
sockaddr_in addr;                                   //声明套接字地址结构变量
addr.sin_family=AF_INET;                            //填充套接字地址结构
addr.sin_port=htons(80);                            //指定数据通信的端口
addr.sin_addr.S_un.S_addr=inet_addr(ip);            //指定服务器 IP 地址
s=::socket(AF_INET,SOCK_STREAM,IPPROTO_TCP);        //创建套接字，并返回其句柄
```

（2）创建服务器套接字

与客户端创建套接字不同。首先，用户需要创建一个专门用于响应客户端连接请求的连接套接字。然后，将该套接字与本地地址绑定在一起。最后，在该套接字上进行监听，监听成功返回新套接字用于收发数据。代码如下：

```
...                                                 //省略部分代码
s=::socket(AF_INET,SOCK_STREAM,IPPROTO_TCP);        //创建连接套接字对象
addr.sin_family=AF_INET;                            //填充套接字地址结构
    addr.sin_port=htons(80);
    addr.sin_addr.S_un.S_addr=inet_addr(str14);
    ::bind(s,(sockaddr*)&addr,sizeof(addr));        //绑定套接字与本地地址
    ::listen(s,1);                                  //监听套接字
```

其中，变量 str14 表示本地 IP 地址。用户可以通过 gethostname()等函数获取本地 IP 地址。代码如下：

```
::gethostname((char*)&name,(int)sizeof(name));      //获得主机名字
```

```
    hostent *p=::gethostbyname((char*)&name);
     in_addr *a=(in_addr*)*p->h_addr_list;              //获得本机 IP 地址
     str14=::inet_ntoa(a[0]);                           //转换为主机字节顺序的 IP
```

当服务器端监听到客户端的连接请求以后，可以调用函数 accept()完成整个连接过程，并返回一个新的套节字。用户收发数据都是通过这个新套接字进行的。代码如下：

```
s1=::accept(s,NULL,NULL);                               //返回数据收发套接字
n=n+1;                                                  //当前客户端的连接数
          str13.Format("有%d 客户已经连接上了",n);      //提示用户当前客户端个数
        this->SetWindowText(str13);
         GetDlgItem(IDC_EDIT1)->GetWindowText((LPTSTR)cs1,10000);
         ::getpeername(s1,(SOCKADDR*)&add,(int*)sizeof(add));
                                                        //获取连接客户端的 IP
         str13+=cs1;
         str13+="\r\n";                                 //添加换行符号
         str13+=::inet_ntoa(add.sin_addr);              //转换为主机字节的 IP
         str13+="登录到聊天室";
         GetDlgItem(IDC_EDIT1)->SetWindowText(str13);
```

通过以上代码，用户可以清楚地看到本地 IP 地址和与服务器连接的客户端 IP 等信息。函数 accept()只能在服务器端进行调用，因为该函数仅用于响应客户端连接请求。

2．连接套接字

在客户端中，套接字创建完成以后，用户需要通过该套接字向服务器发出连接请求。通常，该操作由函数 connect()进行，该函数返回–1，表示失败。否则，表示成功。例如，客户端连接服务器，服务器端 IP 为 218.6.132.5，端口为 80。代码如下：

```
addr.sin_family=AF_INET;
addr.sin_port=htons(80);
addr.sin_addr.S_un.S_addr=inet_addr("218.6.132.5"); //指定服务器 IP 地址
s=::socket(AF_INET,SOCK_STREAM,IPPROTO_TCP);
if(connect(s,(sockaddr*)&addr,sizeof(addr))!=-1)
                                                //向服务器发送连接请求
{
::SendMessage(h,SB_SETTEXT,1,(long)"已经连接上服务器");
                                                //状态栏提示用户已经连接服务器
}
```

当客户端调用 connect()函数向服务器发出连接请求以后，服务器会调用 accept()函数对其进行响应，并返回数据收发套接字。例如，比较简单的服务器响应客户端连接请求。代码如下：

```
...                                     //省略部分代码
if(s1=::accept(s,NULL,NULL)!=NULL)      //响应客户端的连接请求
{
  MessageBox("客户端连接成功！");       //提示用户
}
else
{
MessageBox("客户端连接失败！");
}
```

当用户在客户端界面中，单击“网络设置”按钮后，客户端程序将弹出“设置”对话

框。该“设置”对话框可以根据用户输入的信息连接指定的服务器（默认连接端口为 80）。“设置”对话框，如图 6.3 所示。

在这里，如果运行该程序的机器没有连接网络，则可以使用计算机的回环 IP 地址“127.0.0.1”。当服务器监听并响应客户端的请求后，在服务器端界面上会显示当前连接到服务器的客户端个数，如图 6.4 所示。

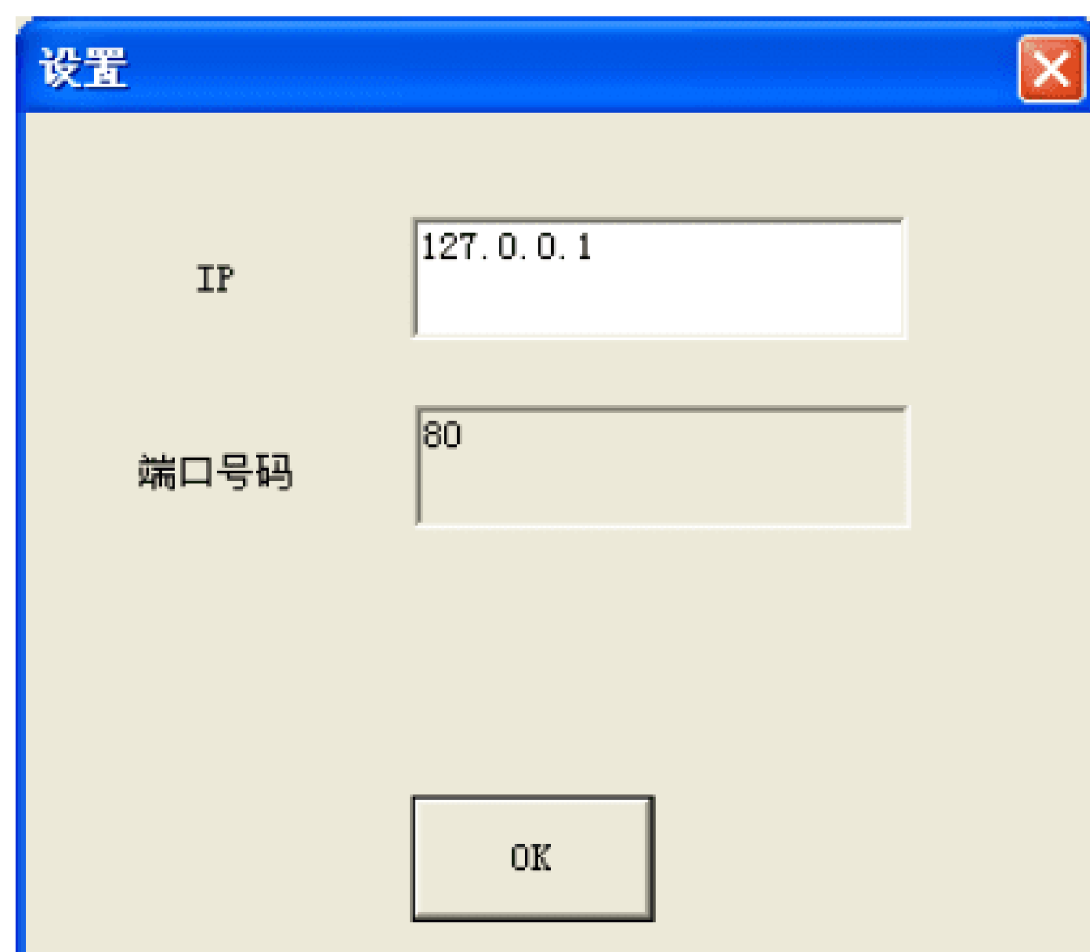


图 6.3　为客户端指定服务器 IP 和端口

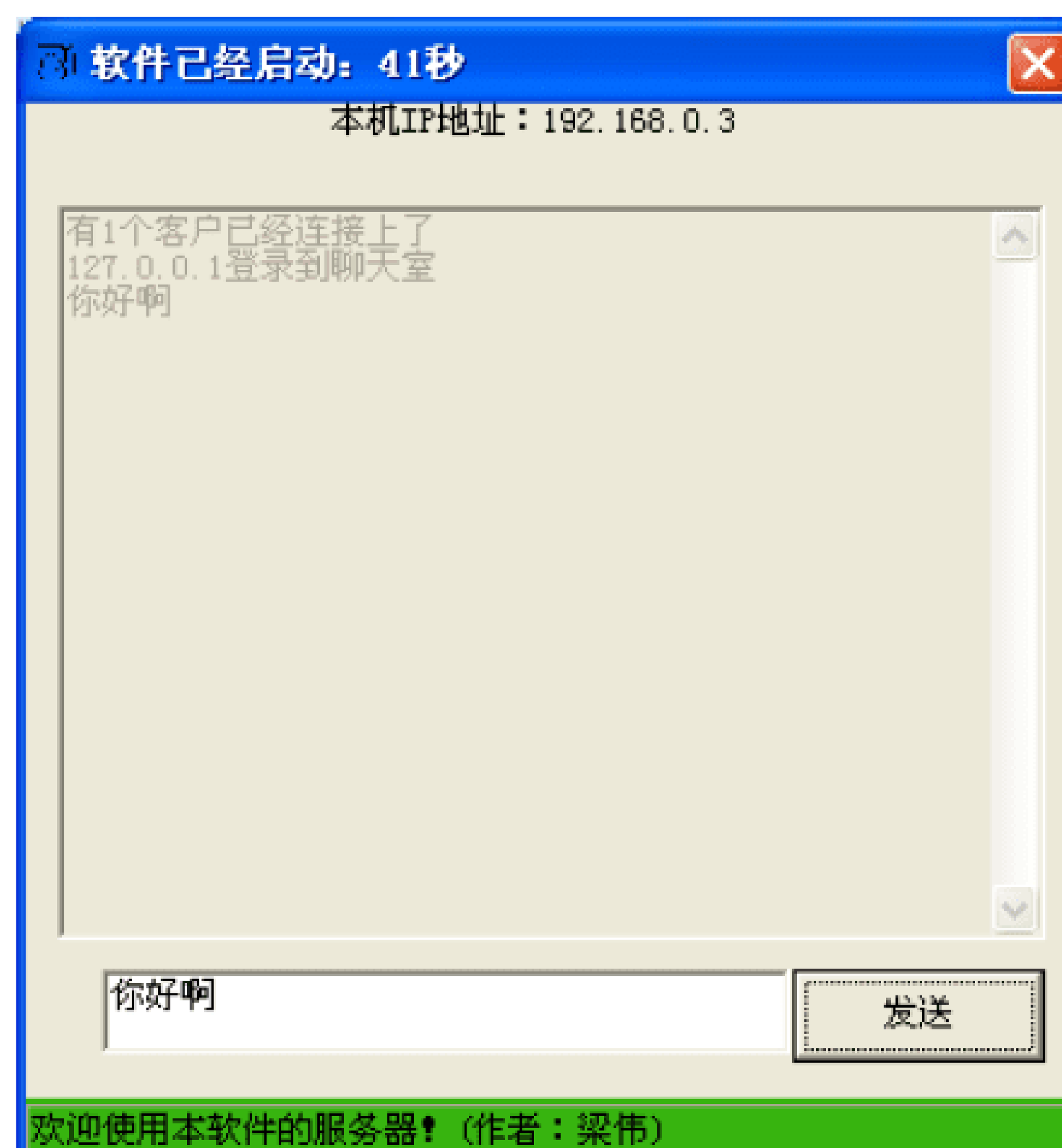


图 6.4　服务器接受连接后显示客户端连接数

用户可以从图 6.4 中看到，服务器端显示了当前连接的客户端数和客户端的 IP 地址等信息。在操作过程中，用户需要特别注意：一定需要先启动服务器，再启动客户端。否则连接不一定会成功。用户从服务器和客户端界面的截图中可以看出，两者的状态栏不但有提示作用，还具有根据时间变化而改变颜色的作用。关于该状态栏的编程应用将在 6.4.4 节内容中详细讲述。

6.1.2　发送接收

现在，用户创建完套接字以后，可以使用该套接字进行数据的发送和接收操作。为了方便用户书写代码，可以通过函数 WSAAsyncSelect()将套接字设置为异步模式，即有相应消息到来时才调用相应的代码。

将客户端套接字设置为异步模式。应该首先自定义消息，然后声明消息响应函数，最后设置异步套接字。首先，在头文件“网络通信 2Dlg.h”中，定义消息 WM_SOCKT。

```
#define WM_SOCKT WM_USER+1
```

然后，再在头文件中声明消息响应函数，保护属性设置为 public。代码如下：

```
afx_msg void Onsockt1(WPARAM wParam,LPARAM lParam);
```

最后，设置套接字为异步模式并在消息响应函数中添加代码。代码如下：

```
...                                                 //省略部分代码
::WSAAsyncSelect(s,this->m_hWnd,WM_SOCKT,FD_READ|FD_ACCEPT);
```

```
                                                              //设置异步套接字
void CMy2Dlg::Onsockt1(WPARAM wParam,LPARAM lParam)    //消息响应函数
 {
     switch(lParam)
 {
case FD_READ:                                                 //处理读取事件
 char s2[100]={0},ssa[20000];
 CString data="";
 sockaddr_in add;                                             //套接字地址结构变量
::memset(&add,0,sizeof(add));
::getpeername(s,(SOCKADDR*)&add,(int*)sizeof(add));    //获取对方 IP 地址
recv(s,s2,100,NULL);                                          //接收数据
GetDlgItem(IDC_EDIT1)->GetWindowText((LPTSTR)ssa,20000);
    data+=(LPTSTR)ssa;                                        //格式化数据
 data+="\r\n";                                                //换行
 data+=::inet_ntoa(add.sin_addr);                             //转换 IP 地址顺序
 data+="对您说：";
 data+=s2;
GetDlgItem(IDC_EDIT1)->SetWindowText(data);                  //设置界面

}
 }
```

对于服务器端而言，其设置异步套接字的方法与客户端一样。代码如下：

```
WSAAsyncSelect(s,this->m_hWnd,WM_SOCK,FD_ACCEPT|FD_READ);
                                                       //设置异步套接字模式
void CMy12Dlg::Onsoc(WPARAM wParam,LPARAM lParam)  //消息响应函数
{   char cs[100],cs1[10000],name[15];
    switch (lParam)
{
      case FD_ACCEPT:                                  //处理连接请求
            {
                s1=::accept(s,NULL,NULL);              //接受客户端的连接请求
         n=n+1;                                        //计数
         str13.Format("有%d 客户已经连接上了",n);      //格式化字符串
        this->SetWindowText(str13);
         GetDlgItem(IDC_EDIT1)->GetWindowText((LPTSTR)cs1,10000);
         ::getpeername(s1,(SOCKADDR*)&add,(int*)sizeof(add));
                                                       //获得连接对方的 IP 地址
         str13+=cs1;
         str13+="\r\n";
         str13+=::inet_ntoa(add.sin_addr);
         str13+="登录到聊天室";
         GetDlgItem(IDC_EDIT1)->SetWindowText(str13);
            }
            break;
        case FD_READ:                                  //处理读取事件
             {
              CString num="";
                     recv(s1,cs,100,NULL);             //接收数据

GetDlgItem(IDC_EDIT1)->GetWindowText((LPTSTR)cs1,10000);

//GetDlgItem(IDC_EDIT2)->GetWindowText((LPTSTR)cs,100);
                     num+=(LPTSTR)cs1;
                       num+="\r\n";
                       num+=::inet_ntoa(add.sin_addr);
```

```
                                            //将 IP 转换为主机顺序
                        num+="对您说：";
                         num+=cs;
                    GetDlgItem(IDC_EDIT1)->SetWindowText(num);}
            break;
}
 }
```

用户将上面的示例程序分别进行编译、运行，可以得到客户端和服务器端应用程序，结果如图 6.5 和图 6.6 所示。

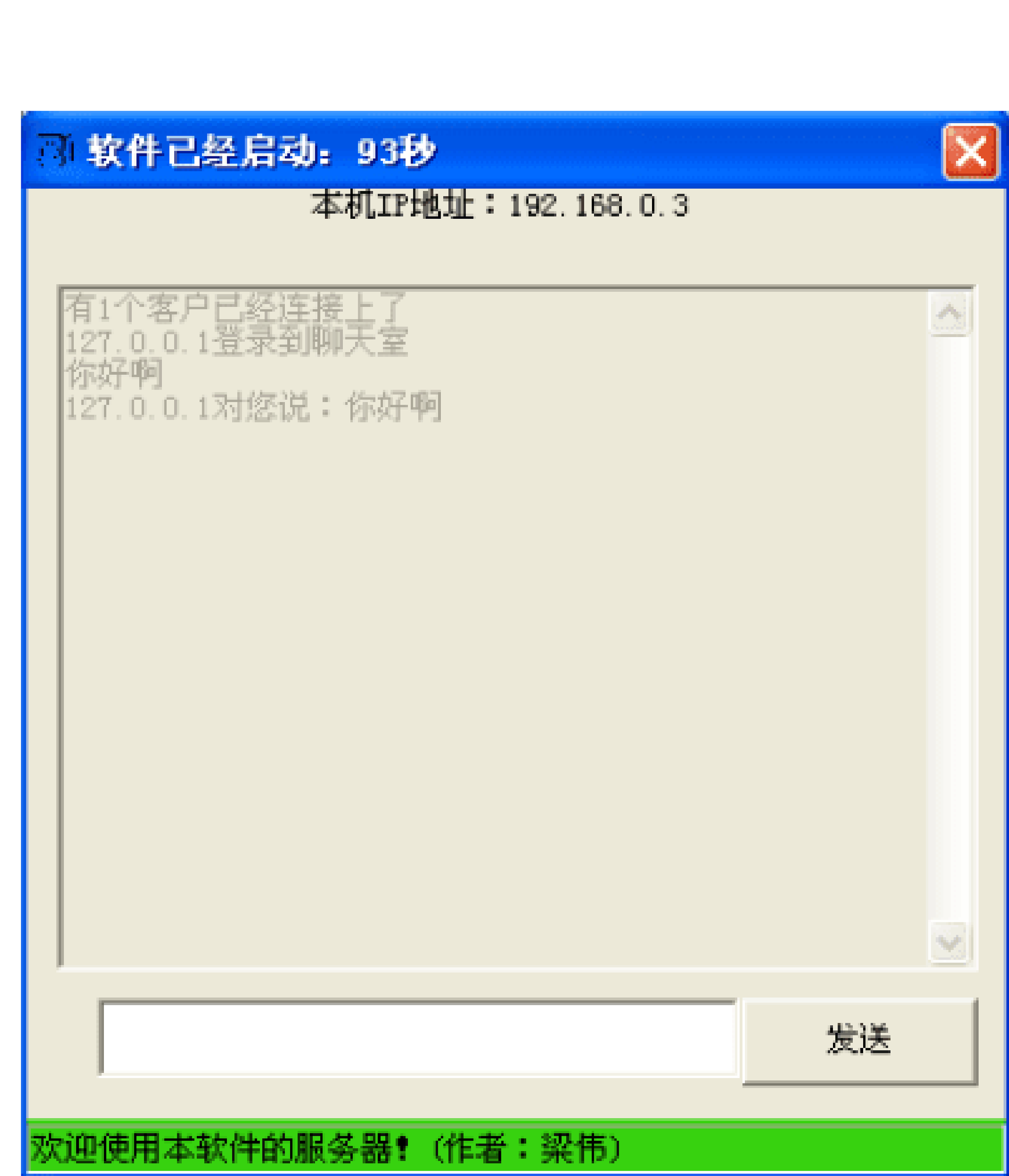


图 6.5　服务器端界面

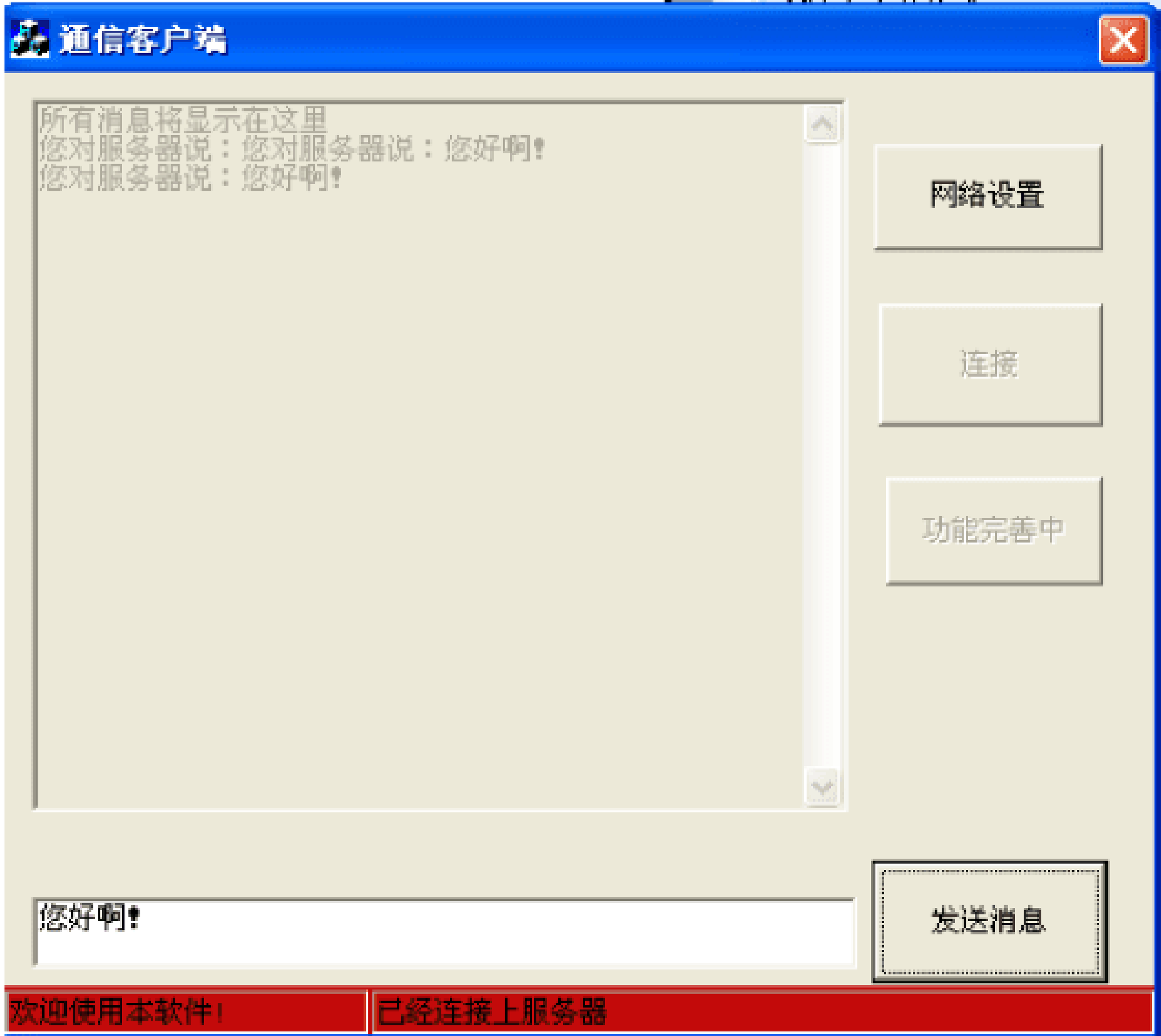


图 6.6　客户端界面

当客户端向服务器发送字符串成功以后，服务器端会显示“str+对您说：+string”。其中，str 表示连接的客户端的 IP 地址，string 表示客户端向服务器发送的字符串。相反亦成立。

注意：在实例中显示的客户端和服务器 IP 地址都是“0.0.0.0”，主要是由于笔者编写该程序时，计算机并未连接网络，程序中所使用的 IP 是计算机的回环 IP“127.0.0.1”。

6.2　发送端程序

本节所涉及到的发送端程序也就是前面所讲到的服务器端。在 VC 中，设计如图 6.2 所示的软件界面。界面设计步骤与前面一样，所以在这里不再赘述。界面中各控件 ID 以及含义如表 6.1 所示。

表 6.1　服务器界面控件ID及含义

ID	含　　义	ID	含　　义
IDS_STATIC	显示本机IP地址	IDC_EDIT2	服务器端发送的数据
IDC_EDIT1	显示连接状态和收发数据	IDC_BUTTON1	发送按钮

6.2.1　创建连接套接字

在服务器编程中，需要用户创建两个套接字对象，分别是连接套接字和数据收发套接字。在本节中，将向用户详细介绍创建服务器连接套接字等具体实现。

首先，在头文件“12Dlg.h”中，声明连接套接字对象，其保护属性修改为 public。代码如下：

```
class CMy12Dlg : public CDialog
{
public:
CMy12Dlg(CWnd* pParent = NULL);                    //构造函数
SOCKET s1;                                         //声明连接套接字对象
...                                                //省略部分代码
}
```

然后，在应用程序初始化函数 OnInitDialog()中添加完整的创建代码。代码如下：

```
BOOL CMy12Dlg::OnInitDialog()
{
    WSADATA data;
DWORD ss=MAKEWORD(2,0);                            //初始化 SOCKET 库
   ::WSAStartup(ss,&data);
   s=::socket(AF_INET,SOCK_STREAM,IPPROTO_TCP);
hh[0]=::LoadIcon(AfxGetApp()->m_hInstance,(char *)IDI_ICON1);
hh[1]=::LoadIcon(AfxGetApp()->m_hInstance,(char *)IDI_ICON2);
//::SetClassLong(m_hWnd,GCL_HCURSOR,(long)cuser);
::gethostname((char*)&name,(int)sizeof(name)); //获得主机名字
hostent *p=::gethostbyname((char*)&name);       //通过主机名获取 IP 地址
 in_addr *a=(in_addr*)*p->h_addr_list;             //取得本机 IP 地址
 str14=::inet_ntoa(a[0]);                          //将 IP 地址转换为主机字节顺序
 str15+="本机 IP 地址：";                           //获得本机 IP 地址
 str15+=str14;
GetDlgItem(IDC_STATIC2)->SetWindowText(str15);
 ::SendMessage(m_hWnd,WM_SETICON,0,(long)hh[0]); //发送 WM_SETICON 消息
 statu=::CreateStatusWindow(WS_CHILD|WS_VISIBLE,"欢迎使用本软件的服务器！（作
者：梁伟)",this->m_hWnd,IDC_123);                   //创建状态栏
::SendMessage(statu,SB_SETBKCOLOR,0,(long)RGB(154,15,15));
                                                   //初始化状态栏颜色
addr.sin_family=AF_INET;
addr.sin_port=htons(80);
addr.sin_addr.S_un.S_addr=inet_addr(str14);      //将 IP 转换为网络字节顺序
::bind(s,(sockaddr*)&addr,sizeof(addr));         //绑定套接字
::listen(s,1);                                     //监听套接字
WSAAsyncSelect(s,this->m_hWnd,WM_SOCK,FD_ACCEPT|FD_READ);
return TRUE;
}
```

最后，当服务器端程序监听到客户端的连接请求以后，调用自定义消息响应函数进行处理相关请求消息。代码如下：

```
void CMy12Dlg::Onsoc(WPARAM wParam,LPARAM lParam)      //消息响应函数
{   char cs[100],cs1[10000],name[15];
switch (lParam)
```

```
{
      case FD_ACCEPT:                                   //处理连接请求
            {
                s1=::accept(s,NULL,NULL);               //接受客户端的连接请求
        n=n+1;                                          //计数
        str13.Format("有%d 客户已经连接上了",n);        //格式化字符串
      this->SetWindowText(str13);
       GetDlgItem(IDC_EDIT1)->GetWindowText((LPTSTR)cs1,10000);
       ::getpeername(s1,(SOCKADDR*)&add,(int*)sizeof(add));
                                                        //获得连接对方的 IP 地址
       str13+=cs1;
       str13+="\r\n";
       str13+=::inet_ntoa(add.sin_addr);
       str13+="登录到聊天室";
       GetDlgItem(IDC_EDIT1)->SetWindowText(str13);
            }
            break;
        case FD_READ:                                   //处理读取事件
              {
               CString num="";
                        recv(s1,cs,100,NULL);           //接收数据

GetDlgItem(IDC_EDIT1)->GetWindowText((LPTSTR)cs1,10000);

//GetDlgItem(IDC_EDIT2)->GetWindowText((LPTSTR)cs,100);
                       num+=(LPTSTR)cs1;
                         num+="\r\n";
                         num+=::inet_ntoa(add.sin_addr);
                                                        //将 IP 转换为主机顺序
                         num+="对您说：";
                          num+=cs;
                    GetDlgItem(IDC_EDIT1)->SetWindowText(num);}
            break;
}
        }
```

通过上面的代码，用户已经创建了服务器连接套接字，并且将该套接字设置为异步模式。用户可以在其响应函数中处理连接和读取事件。

6.2.2　创建发送套接字

在服务器端创建发送套接字，只能通过函数 accept()响应客户端请求，并返回相应的套接字句柄。代码如下：

```
s1=::accept(s,NULL,NULL);                     //接受客户端的连接请求
recv(s1,cs,100,NULL);                         //接收数据
send(s1,cs,100,NULL);                         //发送数据
```

发送套接字 s1 是函数 accept()返回的新套接字对象，如果 s1 不为 NULL，则用户可以在套接字 s1 上接收和发送数据。在本章中，发送套接字主要被用于程序的发送功能。

6.2.3　实现发送功能

根据图 6.2 所示，如果用户需要将数据发送出去，则必须实现发送数据的功能。首先，

用户需要为“发送”按钮添加消息响应函数，如图 6.7 所示。

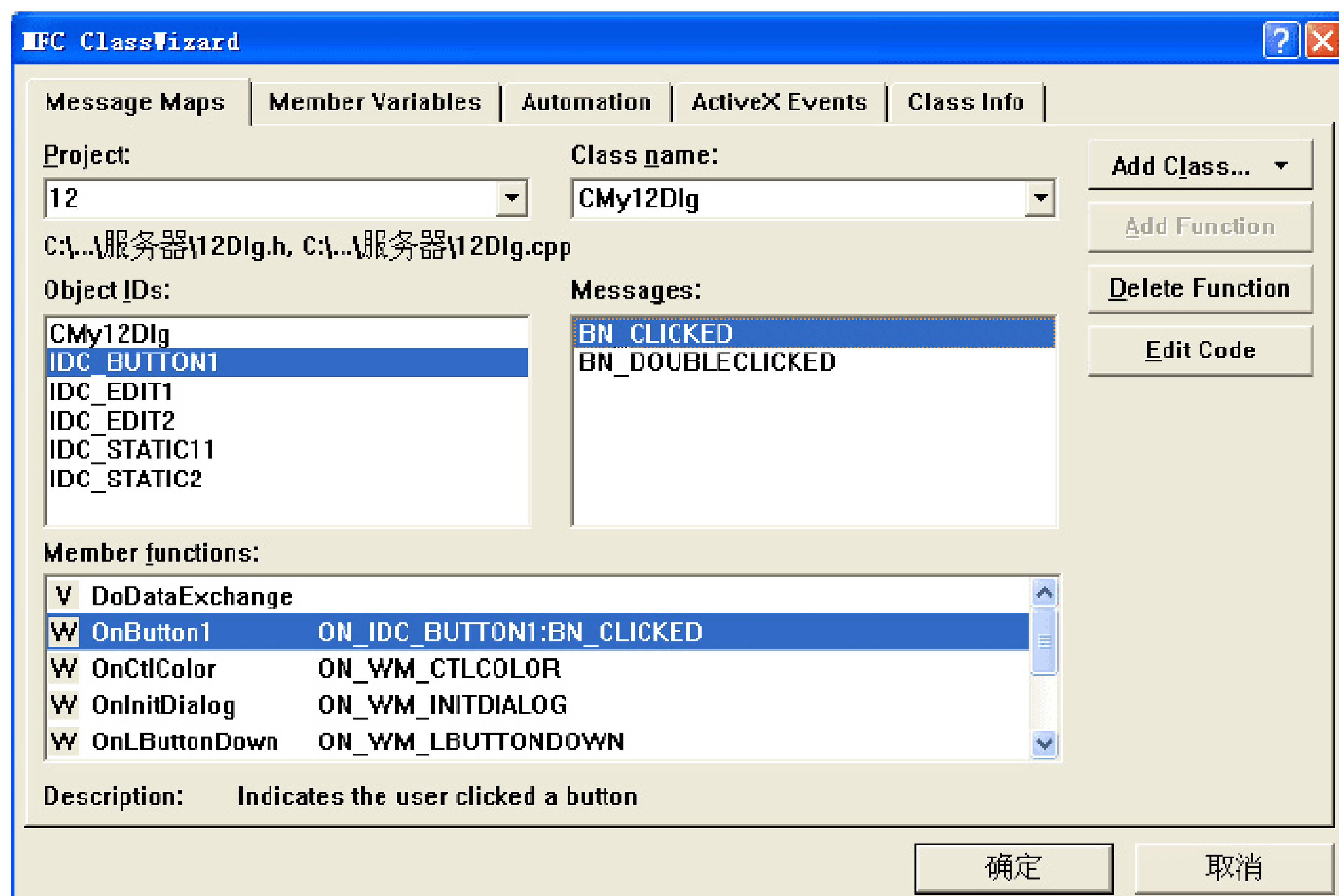


图 6.7　添加“发送”按钮的消息响应函数

然后，用户单击“确定”按钮完成添加消息响应函数。其函数实现如下：

```
void CMy12Dlg::OnButton1()                            //发送按钮消息响应函数
{
// TODO: Add your control notification handler code here
char sever[100];                                      //声明字符数组变量
GetDlgItem(IDC_EDIT2)->GetWindowText((LPSTR)sever,100);
                                                      //获取要发送的数据
CString str="",str1="";                               //声明字符串变量
GetDlgItem(IDC_EDIT1)->GetWindowText(str);            //获取发送框中的数据
if(str!="")                                           //判断发送数据是否为空
{str+="\r\n";}                                        //添加换行符
GetDlgItem(IDC_EDIT2)->GetWindowText(str1);
str+=str1;
    GetDlgItem(IDC_EDIT1)->SetWindowText(str);        //设置程序界面的显示
    send(s1,sever,100,0);                             //发送数据
}
```

以上代码实现了服务器基本的发送功能。并且在将数据发送到客户端的同时，也在服务器本身的界面中显示。这样，可以让用户看见对方发来的消息，也可以看见自己发送给对方的消息，使界面充满了人性化。

在本章实例中，程序仅支持字符串的发送与接收。用户可以在这些功能函数中实现一些字符串与其他格式的数据转换的操作，以便用户在学习套接字编程的基础知识以及基本原理的同时得到其他相关知识的提高。

6.3　接收端程序

接收端程序也就是客户端程序。客户端程序仅仅支持发送和接收数据的功能。在客户

端中，同样具有服务器的功能。例如，监听端口等功能。

6.3.1　监听端口

在客户端中，实现监听端口的功能可以让复杂的原理封装在一个或者几个功能函数中，方便用户编程。微软的 MFC 类库就是一个很好的例子。

客户端实现监听端口功能需要依靠异步套接字，与服务器的监听功能不一样。其原理是自定义一个监听函数，再将客户端的套接字设置为异步模式，套接字的处理事件是 FD_READ。读者需要注意：客户端的监听端口功能是一种逻辑意义上的功能，并非像服务器的监听功能那样时刻监测指定端口。

1．定义监听函数

用户需要为工程添加一个自定义函数作为客户端的监听函数。当然，这个函数可以由用户随意命名。但是为了方便阅读代码和调试代码，所以在本章中，将该函数命名为 Listen。

首先，打开工程的类视图列表。由于工程基于对话框模式，所以在 CMy2Dlg 类中添加 Listen()函数。添加方式是右击 CMy2Dlg 节点，弹出快捷菜单，如图 6.8 所示。

然后，选择 Add Member Funtion 命令，弹出“添加成员函数”对话框，如图 6.9 所示。

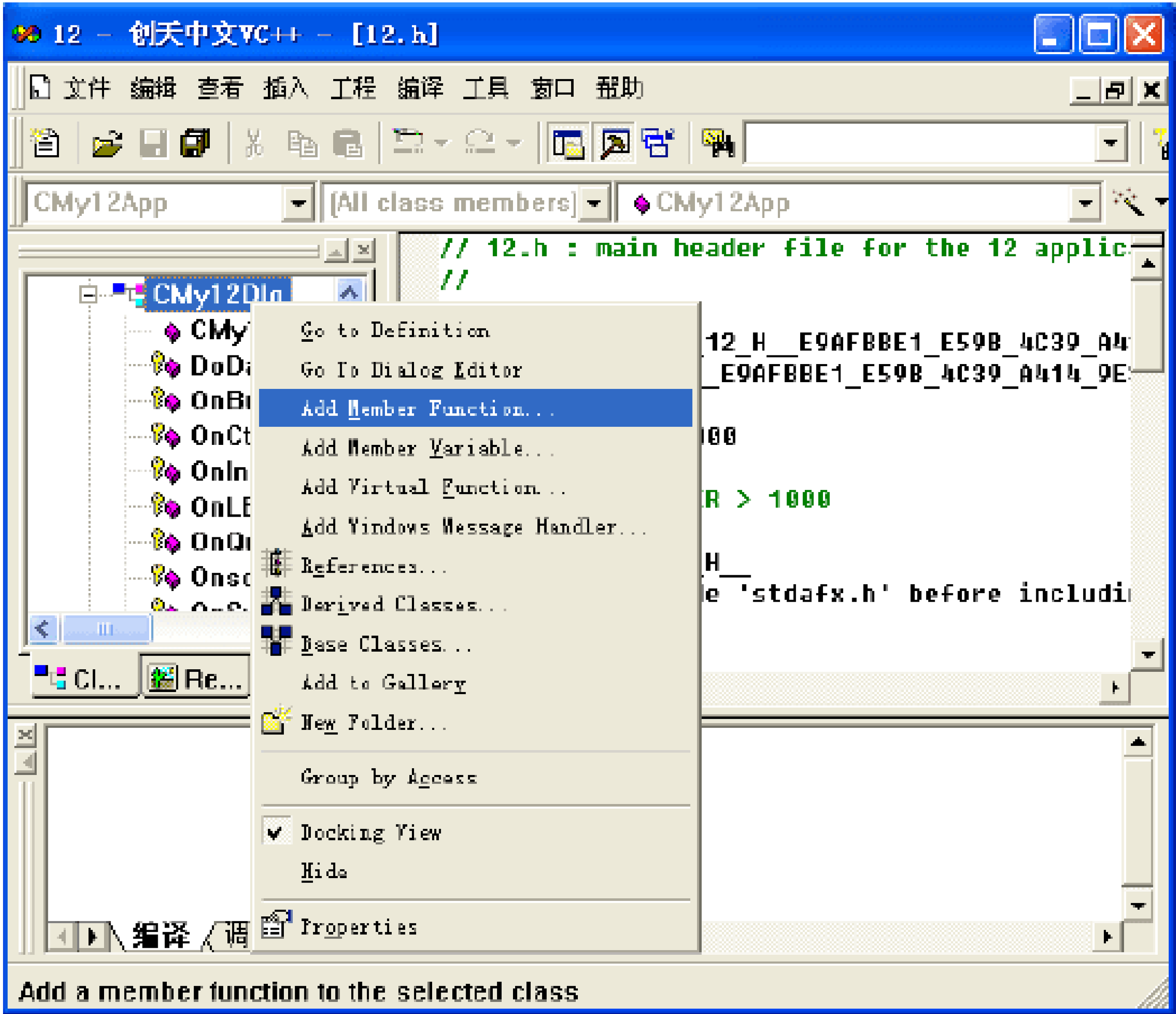


图 6.8　选择添加自定义函数

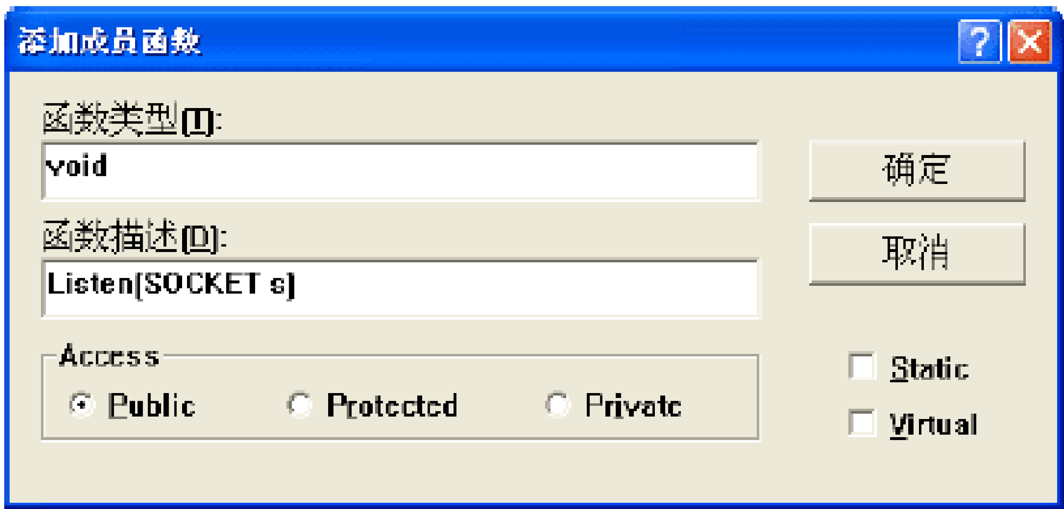


图 6.9 “添加成员函数”对话框

用户在“添加成员函数”对话框中，可以指定自定义函数的一些相关信息，如表 6.2 所示。

表 6.2　自定函数相关信息

函数类型	函数描述	参数类型	参数意义	函数保护属性
void	Listen(SOCKET s)	SOCKET	需要被监听的套接字句柄	Public（公共）

用户将表 6.2 中所示的函数信息填入“添加成员函数”对话框中，然后单击“确定”按钮，编译器会定位到自定义函数处。在函数中将套接字设置为异步模式，代码如下：

```
void CMy2Dlg::Listen(SOCKET s)
{
  ::WSAAsyncSelect(s,this->m_hWnd,WM_LISTENSOCK, FD_READ); //设置异步套接字
...                                                         //省略部分代码
}
```

用户通过上面的代码可以看到自定义函数 Listen()中仅一行代码。作用是将指定套接字设置为异步模式，触发该套接字的事件是 FD_READ，响应消息是 WM_LISTENSOCK。该消息需要在调用前进行自定义。代码如下：

```
#define WM_LISTENSOCK WM_USER+100                           //自定义响应消息
```

在头文件“网络通信 2Dlg.h”中，声明事件消息响应函数 Onlistensockt()。代码如下：

```
asx_msg BOOL Onlistensockt (WPARAM wParam,LPARAM lParam);
```

然后，将自定义消息与响应函数通过消息映射宏联系起来。

```
...                                                  //省略部分代码
BEGIN MESSAGE MAP(CMy2Dlg, CDialog)
//{{AFX_MSG_MAP(CMy2Dlg)
ON_WM_SYSCOMMAND()
ON WM PAINT()
ON_MESSAGE(WM_LISTENSOCK,Onlistensockt)              //建立自定义消息响应
//}}AFX MSG MAP
END_MESSAGE_MAP()
...                                                  //省略部分代码
```

现在，用户已经完成了添加自定义函数 Listen 和异步套接字触发消息以及响应函数。下面，将向用户讲述异步套接字触发消息的响应函数的具体实现方法。代码如下：

```
CString num;                                         //声明全局变量 num
BOOL isrecv;                                         //声明全局变量 isrecv
...                                                  //省略部分代码
BOOL CMy12Dlg:: Onlistensockt (WPARAM wParam,LPARAM lParam)
                                                     //消息响应函数
{   char cs[100],cs1[10000],name[15];
    switch (lParam)
{
        case FD_READ:                                //处理读取事件
              {
                      recv(s1,&cs,100,NULL);         //接收数据

GetDlgItem(IDC EDIT1)->GetWindowText((LPTSTR)cs1,10000);

//GetDlgItem(IDC_EDIT2)->GetWindowText((LPTSTR)cs,100);
                     num+=(LPTSTR)cs1;
                       num+="\r\n";
                        num+=cs;
              if(num!=NULL)
               {
                   isrecv= true;                     //接收成功则赋值 true
}
               else
               {
                   isrecv=false;                     //否则赋值 false
}break;
```

```
default:  break;                                          //处理其他事件
}}
```

在消息响应函数中，如果接收消息成功，则表示端口监听成功并将全局变量 isrecv 赋值为 true，否则监听失败并赋值为 false。然后，用户在自定义监听函数 Listen()中，可以根据变量 isrecv 的值判断监听是否成功。代码如下：

```
void CMy2Dlg::Listen(SOCKET s)
{
  ::WSAAsyncSelect(s,this->m hWnd,WM LISTENSOCK, FD READ);
                                                          //设置异步套接字
     if(isrecv)                                           //判断监听是否成功
       {
         MessageBox("监听端口成功！");                    //提示用户监听成功
          }
     else
       {
         MessageBox("监听端口失败！");                    //否则提示用户监听失败
          }
}
```

2. 调用自定义函数

用户通过以上对监听功能原理的分析，已经知道怎样实现客户端监听端口功能。下面将调用自定义函数 Listen()实现监听功能，代码如下：

```
addr.sin_family=AF_INET;
addr.sin_port=htons(80);
addr.sin addr.S un.S addr=inet addr("218.6.132.5");
s=::socket(AF_INET,SOCK_STREAM,IPPROTO_TCP);   //创建套接字
Listen(s);                                      //调用自定义函数监听套接字
...                                             //省略部分代码
```

本节通过实现自定义监听函数，向用户介绍了在 VC 中添加类成员函数的方法以及客户端监听功能的基本原理。并且通过实现自定义函数，对客户端和服务器端的监听功能进行区别。

6.3.2　接收数据

在 6.3.1 节中，用户实现了自定义监听函数。在该函数中，通过接收数据成功与否来判断端口监听是否成功，这说明接收数据这一功能非常重要。在 Windows 编程中，接收网络数据的函数是 recv()，其原型如下：

```
int recv (SOCKET s,char FAR* buf,int len,int flags);
```

该函数将返回实际接收到的字节数。各参数含义如下：

- ❑ 参数 s 表示通信的套接字句柄。
- ❑ 参数 buf 表示用于接收数据的缓冲区。
- ❑ 参数 len 表示缓冲区的大小。
- ❑ 参数 flags 表示指定的接收模式。一般情况下，将该参数设置为 NULL，表示默认。

使用 recv()函数进行接收数据，代码如下：

```
...                                                    //省略部分代码
addr.sin family=AF INET;
addr.sin port=htons(80);
addr.sin_addr.S_un.S_addr=inet_addr("218.6.132.5");
s=::socket(AF_INET,SOCK_STREAM,IPPROTO_TCP);   //创建套接字
char recv[100]={0};                            //定义并初始化缓冲区
int num=0;                                     //定义接收数据的字节数
num=::recv(s,&recv,100,NULL);                  //调用函数接收数据
if(num==0)                                     //判断接收是否成功
{
   MessageBox("接收数据失败！");                  //提示用户接收失败
    }
else
{
   MessageBox("接收数据成功！");                  //提示用户接收成功
    }
```

如果用户在实际编程中经常需要反复调用函数接收数据，那么应该将接收数据的功能代码封装为自定义函数。这样，不仅减少了在代码中相互调用造成的错误，还方便用户阅读代码。代码如下：

```
void CMy2Dlg::Myrecv(SOCKET s,char *buff,int len,int flags)
{
   addr.sin_family=AF_INET;                    //填充套接字地址结构
addr.sin_port=htons(80);
addr.sin addr.S un.S addr=inet addr("218.6.132.5");
s=::socket(AF_INET,SOCK_STREAM,IPPROTO_TCP);   //创建套接字
int num=0;                                     //定义接收数据的字节数
num=::recv(s,&buff,len, flags);                //调用函数接收数据
if(num==0)                                     //判断接收是否成功
{
   MessageBox("接收数据失败！");                  //提示用户接收失败
    }
else
{
   MessageBox("接收数据成功！");                  //提示用户接收成功
    }
```

在自定义接收函数 Myrecv()中，其参数与 recv()的参数所表示的意义相同，返回类型为 void。如果用户在上节中的异步套接字事件消息响应函数中使用自定义接收函数 Myrecv()，那么具体代码如下：

```
BOOL CMy12Dlg:: Onlistensockt (WPARAM wParam,LPARAM lParam)
                                               //消息响应函数
{   char cs[100],cs1[10000],name[15];
    switch (lParam)
{
        case FD_READ:                          //处理读取事件
               {
                 Myrecv(s1,&cs,100,NULL);      //自定义函数接收数据
                 GetDlgItem(IDC_EDIT1)->GetWindowText((LPSTR)cs1,10000);
                  num+=(LPSTR)cs1;
                 num+="\r\n";
                 num+=cs;
                if(num!=NULL)
                 {
```

```
                isrecv= true;                          //接收成功则赋值 true
}
             else
             {
                isrecv=false;                          //否则赋值 false
}break;
default:  break;                                       //处理其他事件
}}
```

在异步套接字的事件消息响应函数中，调用自定义函数 Myrecv()接收数据不仅提高了代码的运行效率，而且让用户对代码的逻辑性一目了然。因此，用户在以后的编程中，应当尽量将大部分的功能代码封装为一个或者几个主要函数。这样有利于用户养成良好的编程习惯。

6.4　界面美化编程

在本章的前面几节中，主要向用户讲述网络通信器的基本原理和编程技巧等。然而对于一个优秀的即时通信软件而言，还需要美化软件的界面，才能实现软件的人性化设计。因此，软件界面编程也是软件设计的重要步骤，下面本节分别以客户端与服务器端界面编程进行讲解。

6.4.1　界面初始化

在客户端程序界面中，需要用户输入服务器 IP 地址等信息以后，才会提示用户可以连接服务器；或者在用户提供所需信息以后，自动连接服务器。在本实例中，采用提示用户可以连接服务器的方法。客户端启动时的程序界面，如图 6.10 所示。

用户可以在图 6.10 中看到，当客户端界面初始化时，“连接”按钮和“发送消息”按钮均处于禁用状态。如果用户需要连接服务器，则必须单击“网络设置”按钮，弹出“设置”设置对话框，如图 6.11 所示。



图 6.10　客户端启动界面



图 6.11　设置服务器信息

当用户填写好服务器 IP 地址等信息以后，单击 OK 按钮返回客户端主界面。此时“连接”按钮处于可用状态，用户可以通过单击该按钮连接服务器，如图 6.12 所示。

此时，用户单击“连接”按钮开始连接服务器，然后将该按钮设置为禁用状态，并且在状态栏上提示用户“已经连接上服务器”。并且用户可以在编辑框中输入要发送的信息以后，单击“发送消息”按钮进行发送消息，如图 6.13 所示。

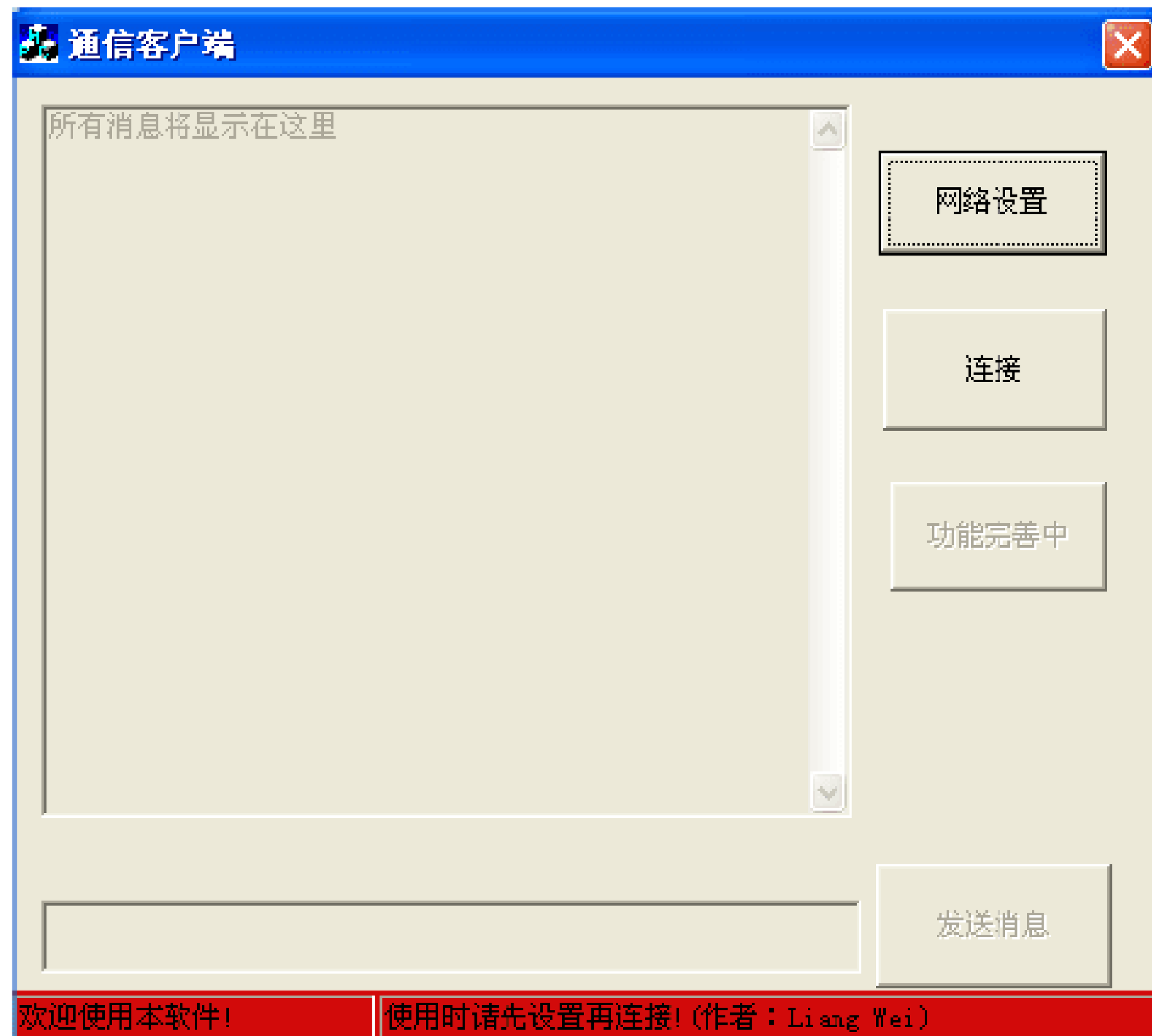


图 6.12　“连接”按钮处于可用状态



图 6.13　连接服务器

注意：*以上过程就是客户端界面从初始化到完全启动所要经历的整个过程，该过程是用户必须做的。*

6.4.2　设置服务器窗口图标

为了使服务器软件运行时，给用户一种运动的感觉，所以在程序中实现图标随时间变化而变化。首先，在工程的资源列表中，插入两幅图标并且分别命名为 IDI_ICON1 和 IDI_ICON2，如图 6.14 所示。

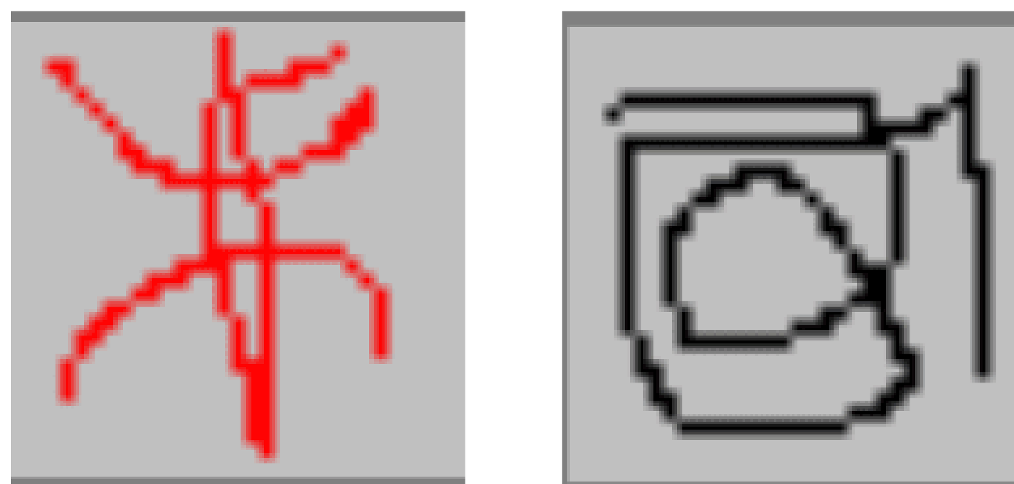


图 6.14　插入的图标资源

然后，在工程的成员函数 OnInitDialog()中载入图 6.14 中的两幅图标资源。代码如下：

```
BOOL CMy12Dlg::OnInitDialog()
{
    ...                                              //省略部分代码
    hh[0]=::LoadIcon(AfxGetApp()->m_hInstance,(char *)IDI_ICON1);
                                                     //载入图标资源
```

```
hh[1]=::LoadIcon(AfxGetApp()->m_hInstance,(char *)IDI_ICON2);
    ::SendMessage(m_hWnd,WM_SETICON,0,(long)hh[0]);      //初始设置窗口图标
    ...                                                  //省略部分代码
}
```

在上面的代码中，变量 hh 是图标资源的句柄变量，主要用于储存读取的图标资源。在本工程中定义句柄变量如下：

```
class CMy12Dlg : public CDialog
{
...                           //省略部分代码
HICON hh[2];                  //定义句柄变量
}
```

注意：该句柄必须定义为 CMy12Dlg 类的成员变量，目的是为了在其他类成员函数中使用该句柄变量。

最后，在工程中添加 WM_TIMER 消息的响应函数，如图 6.15 所示。

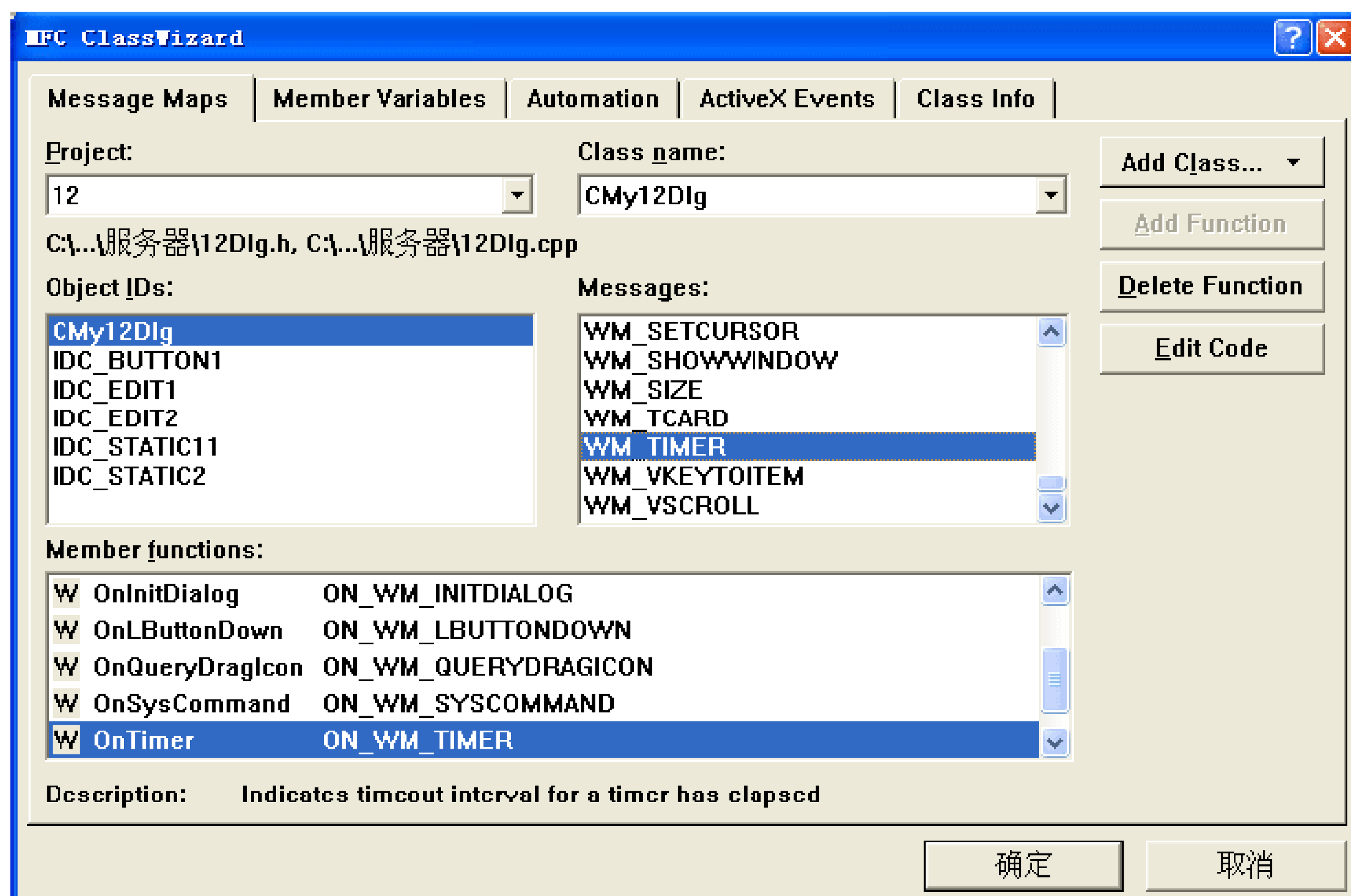


图 6.15　添加消息响应函数

在 WM_TIMER 消息的响应函数 OnTimer()中编写实现轮流显示图标的方法。代码如下：

```
void CMy12Dlg::OnTimer(UINT nIDEvent)
{
if(i<=1)                                      //限制循环次数
{
::SendMessage(m_hWnd,WM_SETICON,0,(long)hh[i++]);
                                              //发送设置图标消息到窗口
}
else                                          //如果限制次数失败，则初始化变量
{
i=0;                                          //为变量赋 0
}
CDialog::OnTimer(nIDEvent);
}
```

在上面的代码中，用户首先对变量 i 进行判断。如果 i 大于了图标资源个数，则初始化 i 为 0，否则将变量加 1，然后向窗口发送消息设置窗口图标。

应用程序通过 WM_TIMER 消息的响应函数就可以让窗口的图标动起来。在程序中，设置程序窗口的图标是通过向窗口发送 WM_SETICON 消息，而该消息的附加值 lParam 为图标资源的句柄。启动窗口后，图标的变化情况如图 6.16 和图 6.17 所示。

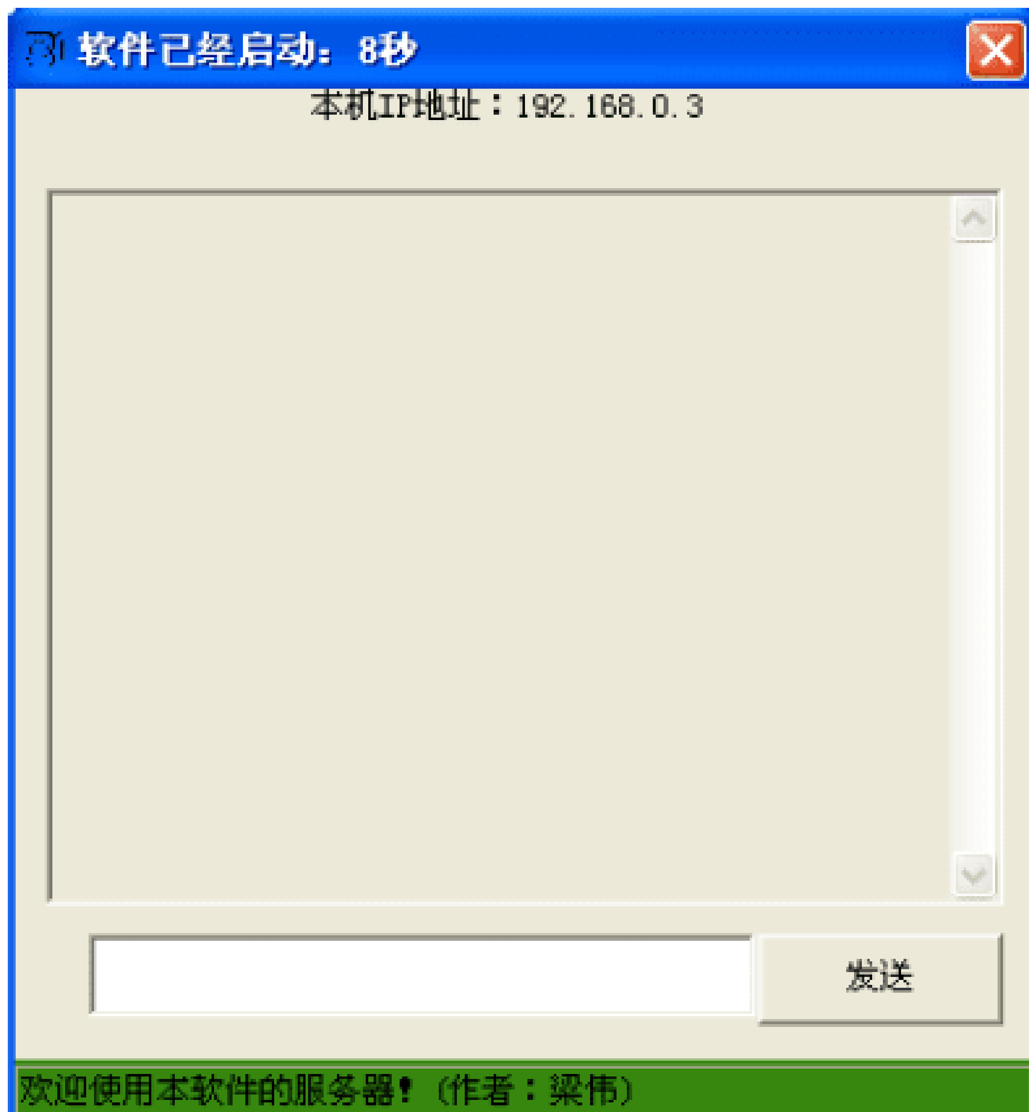


图 6.16　窗口启动 8 秒时的图标

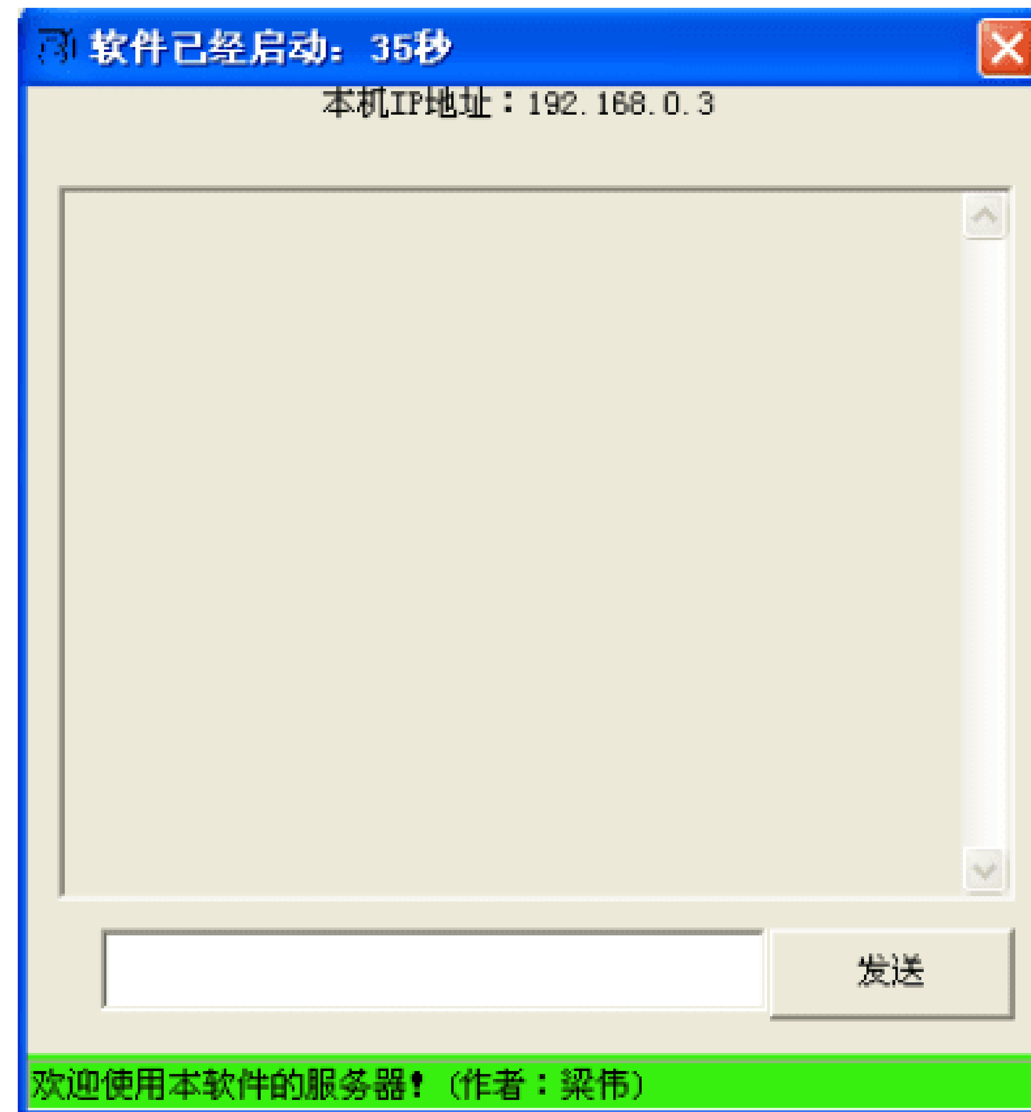


图 6.17　窗口启动 35 秒时的图标

注意：当程序窗口启动时，已经被设置了初始化图标。具体代码见随书光盘。

在图 6.16 和图 6.17 中，读者可以清晰地看到窗口的图标随着事件的变化而变化。实际上，图标的动作主要依赖于计算机中的定时器，其每隔一秒调用一次消息响应函数。而用户则在该消息响应函数中发送 WM_SETICON 消息到窗口设置图标，因为人眼对事物的滞留效应，所以会让用户感觉图标在运动。

6.4.3　显示服务器启动时间

在软件成功启动后，程序该如何告知用户所使用的时间。正如前面所讲到的改变软件的图标一样，时间是变化的，那么该功能的实现还是应该在 WM_TIMER 消息响应函数中。在这里，将向用户介绍一种比较简单的实现该功能的方法。代码如下：

```
int n2=0;                                            //声明全局变量 n2，作为计数变量
...                                                  //省略部分代码
void CMy12Dlg::OnTimer(UINT nIDEvent)
{
  if(i<=1)                                           //限制循环次数
{
    ::SendMessage(m_hWnd,WM_SETICON,0,(long)hh[i++]);
                                                     //发送设置图标消息到窗口
}
else                                                 //如果限制次数失败，则初始化变量
{
i=0;                                                 //为变量赋 0
```

```
}
n2+=1;                                    //每过一秒，n2 自加
str.Format("软件已经启动：%d 秒",n2);     //格式化字符串
this->SetWindowText(str);                 //设置窗口标题
CDialog::OnTimer(nIDEvent);
}
```

代码运行的结果如图 6.17 所示。在本例中，需要向用户说明使用该方法显示程序启动时间，是因为程序中设置了定时器触发时间正好是一秒钟。如果在实际编程中，用户设置的定时器时间不是一秒，那么就需要用户调用 MFC 类库中的函数来实现。下面将向用户介绍实现该功能的几个相关函数。为了在程序中需要获得当前系统的时间可以通过 CTime::GetCurrentTime()函数实现。例如，获得当前系统的时间，代码如下：

```
...                                       //省略部分代码
CTime time1;                              //定义 CTime 类对象
CString str,str1;                         //定义字符串
time1=CTime::GetCurrentTime();            //获取当前系统时间
str=time1.Format("%H:%M:%S");             //格式化输出时间字符串
str1="当前系统的时间是：";                //为字符串赋值
str1+=str;                                //连接两个字符串
MessageBox(str1);                         //提示用户当前的系统时间
...                                       //省略部分代码
```

在 VC 中编译并运行以上代码，其效果如图 6.18 所示。

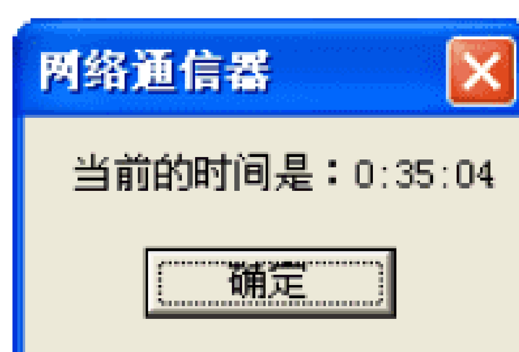


图 6.18　获取当前系统时间

在本实例中，显示程序运行的时间是以秒为单位，所以，用户在本例中格式化时间字符串时只需要获得时间的分秒部分，其余可以舍去。如下所示：

```
int mtime1;                               //声明全局变量
int stime1;
BOOL CMy12Dlg::OnInitDialog()
{
...                                       //省略部分代码
CTime time1;                              //定义 CTime 类对象
CString str,str1;                         //定义字符串
Time1=CTime::GetCurrentTime();            //获取当前系统时间
str=time1.Format("%M);                    //格式化输出时间分钟部分
str1= time1.Format("%S);                  //格式化输出时间秒钟部分
mtime1=atoi(str2);                        //将分钟部分转换为整型数据
stime1=atoi(str3);                        //将秒钟部分转换为整型数据
...                                       //省略部分代码
}
```

上面的代码是获取程序启动时的分钟和秒钟数，要完整的显示程序已运行时间则需要在定时器触发响应函数中动态获取时间。其原理是将动态获取到的时间减去程序启动时的

时间就等于程序已经运行的时间。根据其原理，编程如下：

```
void CMy12Dlg::OnTimer(UINT nIDEvent)
{
   ...                                    //省略部分代码
CTime time2;                              //定义 CTime 类对象
CString str2,str3;                        //定义字符串
time2=CTime::GetCurrentTime();            //获取当前系统时间
str2=time2.Format("%M");                  //格式化输出时间分钟部分
str3= time2.Format("%S");                 //格式化输出时间秒钟部分
int mtime=atoi(str2);                     //将分钟部分转换为整型数据
int stime=atoi(str3);                     //将秒钟部分转换为整型数据
int a;
if((mtime- mtime1)==1)                    //判断分钟数是否相差 1
{
    a=stime+60- stime1;                   //直接使秒数相减得到运行时间
}
else
{
    a= ((mtime- mtime1-1)*60+ stime)- stime1;
                                          //使用分钟的差值乘以 60 与启动时间相减
 }
this->SetWindowText((CString)itoa(a)); //设置窗口标题
...                                       //省略部分代码
}
```

用户根据上面对显示软件运行时间功能的编程描述，结合实际整理代码。在 CMy12Dlg 类的头文件 12Dlg.h 中，添加全局变量的声明。代码如下：

```
int mtime1;                               //声明全局变量
int stime1;
class CMy12Dlg : public CDialog
{
  ...                                     //省略部分代码
}
```

然后，根据程序启动时所获取时间和定时器触发消息时的时间来确定软件运行的时间。代码如下：

```
BOOL CMy12Dlg::OnInitDialog()             //程序初始化函数
{
CTime time1;                              //定义 CTime 类对象
CString str,str1;                         //定义字符串
Time1=CTime::GetCurrentTime();            //获取当前系统时间
str=time1.Format("%M");                   //格式化输出时间分钟部分
str1= time1.Format("%S");                 //格式化输出时间秒钟部分
mtime1=atoi(str2);                        //将分钟部分转换为整型数据
stime1=atoi(str3);                        //将秒钟部分转换为整型数据
}
void CMy12Dlg::OnTimer(UINT nIDEvent)     //定时器触发消息函数
{
CTime time2;                              //定义 CTime 类对象
CString str2,str3;                        //定义字符串
time2=CTime::GetCurrentTime();            //获取当前系统时间
str2=time2.Format("%M");                  //格式化输出时间分钟部分
str3= time2.Format("%S");                 //格式化输出时间秒钟部分
```

```
int mtime=atoi(str2);                        //将分钟部分转换为整型数据
int stime=atoi(str3);                        //将秒钟部分转换为整型数据
int a;
if((mtime- mtime1)==1)                       //判断分钟数是否相差 1
{
   a=stime+60- stime1;                       //直接使秒数相减得到运行时间
}
else
{
   a= ((mtime- mtime1-1)*60+ stime)- stime1;
                                             //使用分钟的差值乘以 60 与启动时间相减
 }
this->SetWindowText((CString)itoa(a)); //设置窗口标题
}
void CMy12Dlg::OnClose()                     //关闭函数
{
    KillTimer(1);                            //关闭定时器
    CDialog::OnClose();
}
```

用户通过以上代码便可以很轻松地实现显示软件已运行时间。但是根据需要，用户可以更改以上代码以达到现实需要，具体代码请参考随书光盘。

6.4.4　服务器状态栏编程

状态栏在软件中可以作为一个提示牌显示一些提示信息给用户，或者是显示帮助。在图 6.12 中，用户可以发现程序界面中不但有可以变化的图标和时间，还有可以改变颜色的状态栏。状态栏编程在软件设计中也是很重要的一部分。

首先，创建状态栏需要使用 Win32 API 函数 CreateStatusWindow()。其原型如下：

```
HWND CreateStatusWindow(
   LONG style,
   LPCTSTR lpszText,
   HWND hwndParent,
   UINT wID
);
```

该函数创建状态栏成功则返回该状态栏的句柄，否则返回 NULL。其参数如下：

- 参数 style 表示创建的状态栏窗口样式，部分取值如表 6.3 所示。

表 6.3　窗口样式的部分取值

取　　值	意　　义
WS_POPUP	创建弹出窗口
WS_MINIMIZEBOX	具有最小化按钮
WS_CAPTION	具有标题栏
WS_MAXIMIZEBOX	具有最大化按钮
WS_VISIBLE	创建的窗口可见
WS_SYSMENU	具有系统菜单的窗口
WS_CHILD	创建子窗口
WS_MINIMIZE	程序窗口启动时为最小化状态
WS_MAXIMIZE	程序窗口以屏幕大小启动
WS_HSCROLL/ WS_VSCROLL	程序窗口带有水平或者垂直滚动条

- 参数 lpszText 表示在状态栏上的文字。
- 参数 hwndParent 表示新创建的状态栏的父窗口句柄。
- 参数 wID 表示状态栏的 ID，该 ID 必须事先进行声明。

用户根据表 6.3 中的窗口样式和状态栏在程序窗口中的位置可以知道，创建状态栏时需要为其指定的窗口样式为“WS_CHILD|WS_VISIBLE”。代码如下：

```
BOOL CMy12Dlg::OnInitDialog()
{
statu=::CreateStatusWindow(WS_CHILD|WS_VISIBLE, "欢迎使用本软件的服务器！(作者：Liangwei)",this->m_hWnd,IDC_123);          //创建状态栏
...                                               //省略部分代码
}
```

在代码中。状态栏句柄 statu 和状态栏的 ID 都需要事先在相关文件中进行声明。例如，在头文件中声明状态栏句柄。

```
class CMy12Dlg : public CDialog
{
protected:
HWND statu;                                       //声明状态栏句柄
}
```

在头文件 Resource.h 中定义状态栏的 ID。定义代码如下：

```
#define IDC_123                    1009
```

通过以上的操作和代码编写，已经在程序窗口的界面上添加了一个状态栏，如图 6.19 所示。

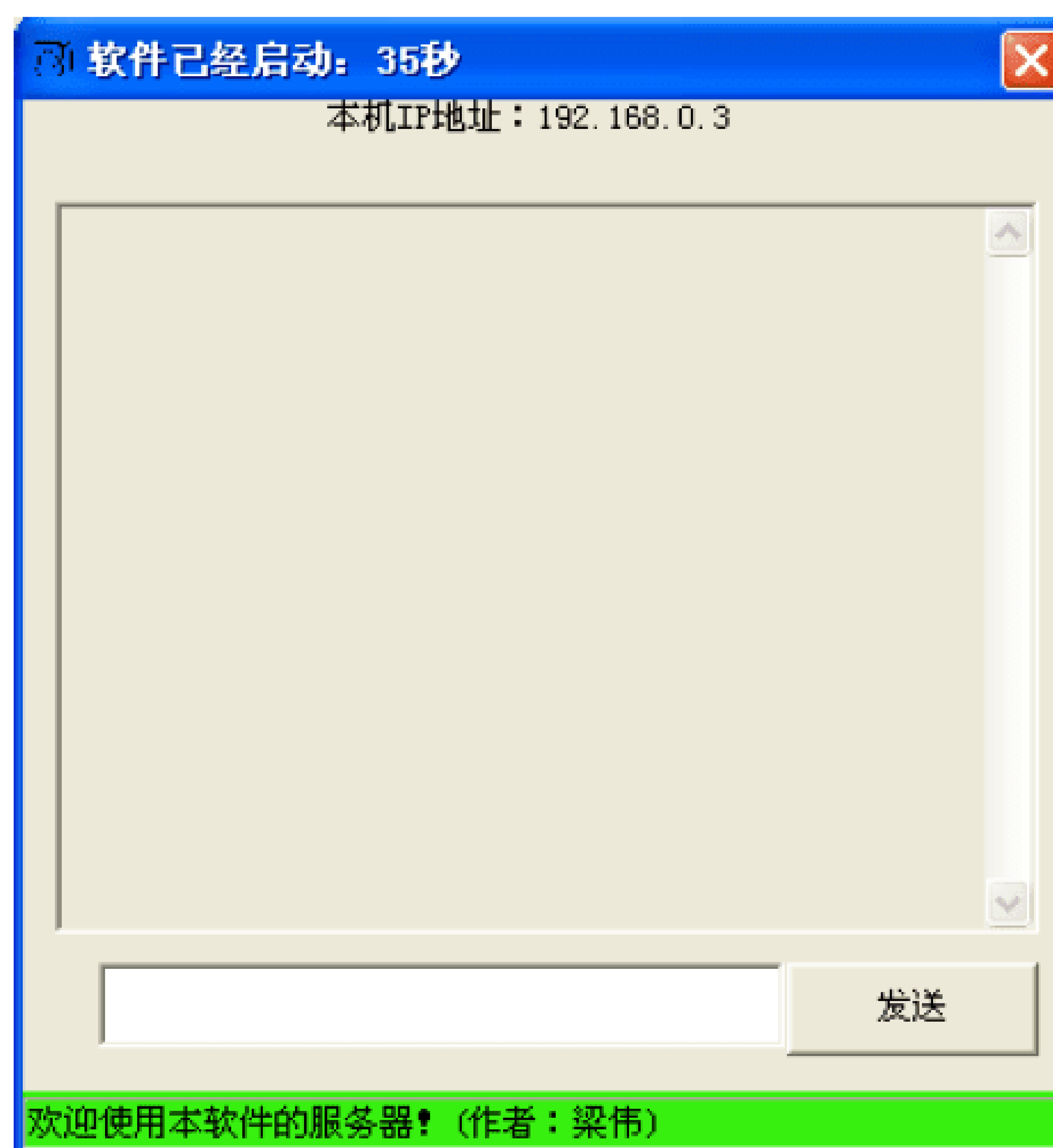


图 6.19　在窗口界面上添加状态栏

然后，在程序运行时根据时间的变化而变化状态栏的颜色。改变状态栏的颜色可以向状态栏窗口发送消息 SB_SETBKCOLOR，通过该消息的附加参数 lParam 指定颜色值。例如：

```
::SendMessage(statu,SB_SETBKCOLOR,0,(long)RGB(154,15,15);
```

当窗口显示时，状态栏必须显示初始颜色值，那么在窗口初始化函数 OnInitDialog()中添加以下代码：

```
BOOL CMy12Dlg::OnInitDialog()
{
    ...                                                    //省略部分代码
::SendMessage(statu,SB SETBKCOLOR,0,(long)RGB(255,0,0);
                                                           //设置状态栏初始颜色值为红色
...                                                        //省略部分代码
}
```

与改变窗口图标等实现原理一样，改变状态栏颜色同样也是在定时器触发消息响应函数中进行实现。但是需要用户注意：RGB 颜色值的范围在 0～255 之间，这是因为在计算机系统中每个颜色值分量由 8 个二进制数组成。

在定时器触发的消息响应函数中，编写代码实现变换颜色的状态栏。代码如下：

```
int i=0;                                                   //定义并初始化全局变量
void CMy12Dlg::OnTimer(UINT nIDEvent)                      //定时器触发消息函数
{
     i+=15;                                                //变量自加 15
    if(i>255)                                              //如果 i 值大于 255，则赋值 15
     {
i=15;                                                      //赋值 15
}
else
{
::SendMessage(statu,SB SETBKCOLOR,0,(long)RGB(i,0,15));
                                                           //否则发送消息设置状态栏颜色值
this->SetWindowText("程序正在改变颜色！");                 //提示用户程序正在改变颜色
}
}
```

将以上代码在 VC 中进行编译，观察其窗口状态栏的颜色改变情况如图 6.20 和图 6.21 所示。在图 6.20 中，所示为程序窗口启动时的状态栏颜色。在图 6.21 中，所示为程序窗口运行后的状态栏颜色。



图 6.20　窗口启动时的状态栏颜色

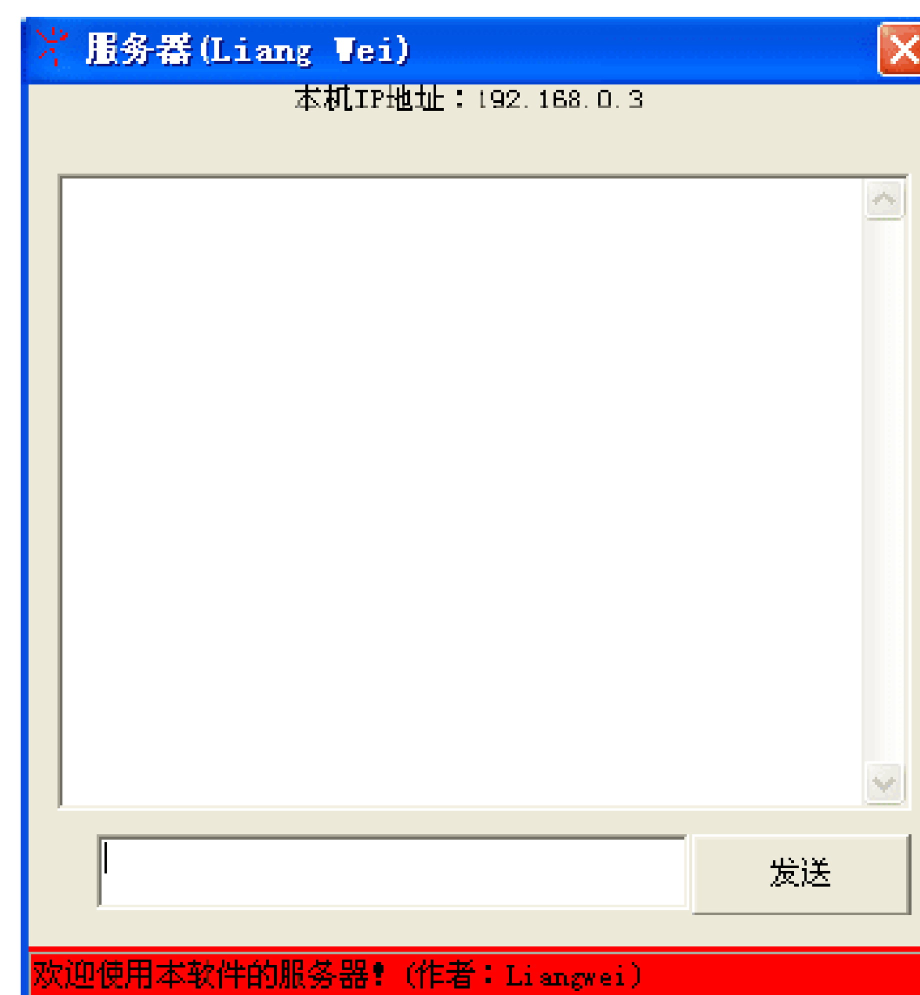


图 6.21　窗口运行后的状态栏颜色

根据两幅截图可以知道，程序界面在不同时刻的颜色不同，用户可以打开随书光盘中的程序自行编译以后查看效果。用户也可以试一试修改颜色值，看看修改后的效果。

用户通过对图 6.20 和图 6.21 进行比较，可以很明显地发现程序窗口的状态栏颜色发生了很大变化。这样做不仅使软件窗口看起来非常美观和人性化，还会让使用软件的人保持一种动态感。

注意： 用户在以后的界面编程中，软件界面的美化编程也是程序设计中重要的一部分。本例中的具体代码请用户参考随书光盘。

6.5　小　　结

本章第一部分中，主要向用户讲述了网络通信器的通信原理，根据该原理进行 MFC 程序的设计。在 VC 编译环境下，向用户讲述了网络通信器的服务器端和客户端编程的方法以及流程。在客户端编程中，使用自定义函数对监听端口的功能进行了封装，用实例向用户介绍了客户端监听端口功能的实现原理。

在第二部分中，主要向用户讲解在 VC 的界面编辑器中布置软件的界面，并且编程实现各个按钮的功能。在这一部分中，通过实现变化的图标、软件运行时间显示和可变色的状态栏等编程向用户讲述界面各部分的功能以及实现方法。

通过对本章的学习，用户将对网络通信器的设计及实现方法有更加深入的理解。同时，用户可以将本章中的实例程序进行修改以达到自我检测和学习的目的。

第 7 章　邮件收发器

邮件收发器的作用是在本地计算机和远程计算机之间传送电子信件以及接收电子信件等操作。用户平时所用的 E-mail 就是一种邮件收发器。通常情况下，E-mail 由发送者将电子信件发送到邮件服务器（SMTP）中，再由 SMTP 服务器将该邮件发送到 POP3（接收邮件）服务器中，邮件接收者通过账户和口令再从 POP3 服务器中获取信件。在本章中，将向用户介绍邮件收发器的原理以及开发过程。

7.1　调用 Windows 自带的邮件发送程序

一般情况下，用户所使用的 Windows 操作系统中都带有默认的邮件发送程序。通过该邮件发送程序，用户可以将邮件发送到任何目的地址。这种方法比较简单适用，所以很受大部分用户欢迎。用户可以在操作系统中，使用操作系统命令打开邮件程序。如果用户需要在自己的程序中调用系统自带的邮件程序，那么需要使用函数 CreateProcess()或者 ShellExecute()进行调用。下面将分别介绍这两种方法。

7.1.1　调用 Windows 进程

在 Windows 操作系统中，所有的程序都是以进程为单位运行。本节中所讲述的调用邮件发送程序就是通过调用相应的 Windows 进程实现的。调用该 Windows 进程所使用的命令是“mailto:+string”，其中，string 表示邮件发送的目的地址。例如，用户需要将邮件发送到邮件地址为 lymlrl@163.com 的邮箱中，使用的命令是“mailto:lymlrl@163.com”。

首先，在 Windows 系统界面下选择“开始”|“运行”命令（如图 7.1 所示），弹出“运行”对话框，如图 7.2 所示。



图 7.1　打开运行对话框



图 7.2　“运行”对话框

然后，在运行对话框中输入命令“mailto:lymlrl@163.com”，可以打开 Windows 自带的邮件发送程序进行邮件发送，如图 7.3 所示。

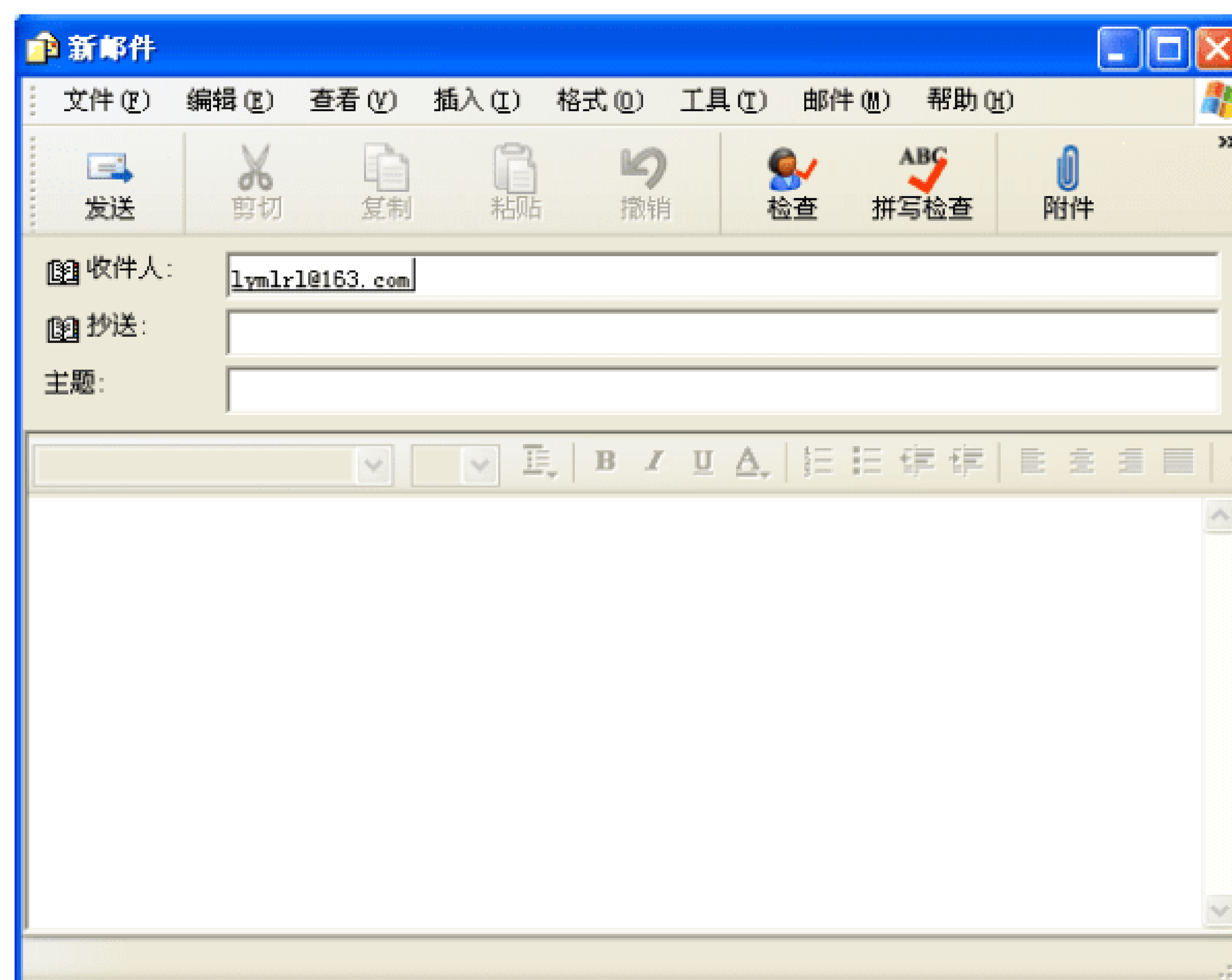


图 7.3　Windows 邮件收发器

以上过程是用户通过 Windows 命令调用邮件收发器必须做的。实际上，除了这种方法，用户还可以在程序中通过函数调用 Windows 邮件收发器。此种方法将在 7.1.2 节中进行讲解。

7.1.2　CreateProcess()函数

在 VC 中编程，MFC 类库已经提供了几个库函数用于调用 Windows 的外部程序，包括邮件收发程序。在本节中，将向用户介绍其中的两个函数 CreateProcess() 和 ShellExecute()。

1．使用CreateProcess()函数

CreateProcess()函数可以创建 Windows 进程，同时也可以调用已经存在的进程。该函数的原型如下：

```
BOOL CreateProcess(
  LPCTSTR lpApplicationName,
  LPTSTR lpCommandLine,
LPSECURITY ATTRIBUTES lpProcessAttributes,
  LPSECURITY_ATTRIBUTES lpThreadAttributes,
  BOOL bInheritHandles,
  DWORD dwCreationFlags,
  LPVOID lpEnvironment,
  LPCTSTR lpCurrentDirectory,
  LPSTARTUPINFO lpStartupInfo,
  LPPROCESS_INFORMATION lpProcessInformation
);
```

该函数创建进程成功则返回 true，否则返回 false。其参数意义如下：

- 参数 lpApplicationName 表示可执行文件的名字。用户指定该参数后，该函数会在当前路径下搜索可执行文件，但不会按照系统的搜索路径进行搜索。

注意：使用该参数时，需要加上扩展名，因为系统不会自动为其添加“.exe”后缀名。

- 参数 lpCommandLine 表示将要传递到新进程的命令行字符串。使用该参数时，该函数会自动为其添加后缀名“.exe”。如果参数字符串没有指定所在路径，那么该函数则会按照系统的搜索路径进行搜索文件。
- 参数 bInheritHandles 表示该进程创建的子进程是否能继承父进程的对象句柄。
- 参数 lpStartupInfo 指向结构体 STARTUPINFO 的指针变量。该结构体的声明如下：

```
typedef struct _STARTUPINFO {
    DWORD   cb;                 //表示该结构体的大小
    LPTSTR  lpReserved;         //保留，必须将该参数初始化为 NULL
    LPTSTR  lpDesktop;          /* 用于标识启动应用程序所在的桌面的名字。如果该桌面存
                                在，新进程便与指定的桌面相关联。如果桌面不存在，便创建
                                一个带有默认属性的桌面，并使用为新进程指定的名字。如果
                                lpDesktop 是 NULL（这是最常见的情况），那么该进程将与
                                当前桌面相关联 */
    LPTSTR  lpTitle;            //设置控制台程序的名称
    DWORD   dwX;                //设置应用程序窗口的 X 坐标
    DWORD   dwY;                //设置应用程序窗口的 Y 坐标
    DWORD   dwXSize;            //设置应用程序窗口的横向大小
    DWORD   dwYSize;            //设置应用程序窗口的纵向大小
    DWORD   dwXCountChars;      //以字符为单位设置应用程序窗口的 X 坐标
    DWORD   dwYCountChars;      //以字符为单位设置应用程序窗口的 Y 坐标
    DWORD   dwFillAttribute;    //设置应用程序窗口所使用的背景色等
    DWORD   dwFlags;            //表示创建窗口的标志
    WORD    wShowWindow;        //是否显示应用程序窗口
    WORD    cbReserved2;        //保留，将该参数必须设置为 0
    LPBYTE  lpReserved2;        //保留，将该参数必须设置为 0
    HANDLE  hStdInput;          //设置控制台程序的输入输出缓存句柄
    HANDLE  hStdOutput;
    HANDLE  hStdError;          //错误输出句柄
} STARTUPINFO, *LPSTARTUPINFO;
```

该结构体主要用于保存新创建进程的窗口信息，如窗口的大小或窗口的显示方式等。其中，参数 dwFlags 标识了窗口创建成功以后，在显示之前以何种方式进行显示。其取值如表 7.1 所示。

表 7.1　程序窗口显示标志取值

取　　值	含　　义
STARTF_USESIZE	使用 dwXSize 和 dwYSize 成员
STARTF_USESHOWWINDOW	使用 wShowWindow 成员
STARTF_USEPOSITION	使用 dwX 和 dwY 成员
STARTF_USECOUNTCHARS	使用 dwXCountChars 和 dwYCountChars 成员
STARTF_USEFILLATTRIBUTE	使用 dwFillAttribute 成员
STARTF_USESTDHANDLES	使用 hStdInput、hStdOutput、hStdError 成员
STARTF_RUN_FULLSCREEN	以全屏方式启动程序

注意： 在表 7.1 中所示的程序窗口显示标志的作用仅仅是为了控制相应的成员变量是否有效而已。例如，用户在程序中，需要使用到该结构体中的 dwFillAttribute 成员。那么，用户必须将参数 dwFlags 取值为 STARTF_USEFILLATTRIBUTE。否则，该成员变量将无效。

- 参数 lpProcessInformation 是指向结构体 PROCESS_INFORMATION 的指针变量。该结构体声明如下：

```
typedef struct _PROCESS_INFORMATION
 {
   HANDLE hProcess;                    //进程句柄
   HANDLE hThread;                     //线程句柄
   DWORD dwProcessId;                  //进程 ID
   DWORD dwThreadId;                   //线程 ID
} PROCESS_INFORMATION;
```

该结构体主要用于保存进程的相关信息。其他参数均可以默认设置为 NULL。例如，调用操作系统的记事本程序。代码如下：

```
...                                    //省略部分代码
STARTUPINFO si={sizeof(si)};           //定义结构体变量
PROCESS_INFORMATION pi;                //定义结构体对象
CString *str="notepad";                //记事本名称
CreateProcess(NULL,str,NULL,NULL,false,NULL,NULL,NULL,&si,&pi);
                                       //调用函数打开记事本程序
...                                    //省略部分代码
```

同样的道理，用户在本例中，也可以使用函数 CreateProcess()调用邮件收发程序。代码如下：

```
...                                    //省略部分代码
STARTUPINFO si={sizeof(si)};           //定义结构体变量
PROCESS INFORMATION pi;
CString *str="mailto:lymlrl@163.com";  //打开邮件程序的系统命令
CreateProcess(NULL,str,NULL,NULL,false,NULL,NULL,NULL,&si,&pi);
                                       //调用函数打开记事本程序
...                                    //省略部分代码
```

2. 使用ShellExecute()函数

在 MFC 编程中，除了函数 CreateProcess()以外，还可以调用函数 ShellExecute()实现相同的功能。该函数原型如下：

```
HINSTANCE ShellExecute(
   HWND hwnd,                          //父窗口句柄
   LPCTSTR lpOperation,                //将要进行的操作形式
   LPCTSTR lpFile,                     //目录文件名称或文件路径
   LPCTSTR lpParameters,               //传递的参数
   LPCTSTR lpDirectory,                //一般为 NULL
   INT nShowCmd                        //显示方式
);
```

该函数执行成功会返回调用程序的应用程序指针，否则返回错误代码。部分错误代码如表 7.2 所示。

表 7.2　部分错误代码

错 误 代 码	意　　义
ERROR_FILE_NOT_FOUND	找不到相应文件
ERROR_PATH_NOT_FOUND	找不到所需路径
ERROR_PATH_NOT_FOUND	无效的.exe 文件
SE_ERR_ASSOCINCOMPLETE	无效的文件名
0	操作系统的内存溢出

该函数的各个参数说明以及意义已经在第 5 章的 5.4.3 节中讲述，在这里不再赘述，请用户复习前面的知识点。使用该函数调用操作系统自带的邮件发送程序，代码如下：

```
#include <stdio.h>                              //调用相关头文件
#include <windows.h>
main()                                          //主函数
{
  int i=0;                                      //定义循环变量
  char ch;                                      //定义字符，用于获取用户输入
printf("确认打开邮件收发程序!(Y/N)\n");          //提示用户
scanf(&ch);                                     //输入指令
if(ch && 'Y')                                   //判断输入指令
{
    printf("邮件收发程序正在打开！请稍候……\n");   //提示用户
    while(i<=10000000)                          //循环，模拟计算机工作
{
    i+=1;
}
ShellExecute(NULL,NULL,"mailto:lymlrl@163.com",
NULL,NULL,SW_SHOW);                             //调用函数启动邮件收发程序
printf("邮件收发程序已经打开，请使用！\n");
}
else
{    printf("谢谢使用!\n");    }}
```

以上代码是使用 C 语言编写，并且使用命令行窗口界面，目的是为了让用户了解整个调用过程。在随书光盘的第 7 章中附有代码，请用户自行参考。此段代码在 VC 中编译后的结果，如图 7.4 所示。用户在运行界面 1 中输入字符 Y 或 y，然后按下 Enter 键。程序提示邮件程序正在打开，当邮件程序打开以后，实例程序会提示已经打开邮件程序，如图 7.5 所示。

注意：在程序中为了模拟计算机的工作，所以笔者使用了 while 循环产生时间差，仅仅是为了让用户重复了解该调用过程。在实际编程中，不提倡使用该方法产生时间差，因为这种方法很危险，容易造成系统的崩溃。通常，使用多线程编程的方法比较安全，也是笔者极力推荐的一种方法。该类方法将在后面的相关章节中讲述。

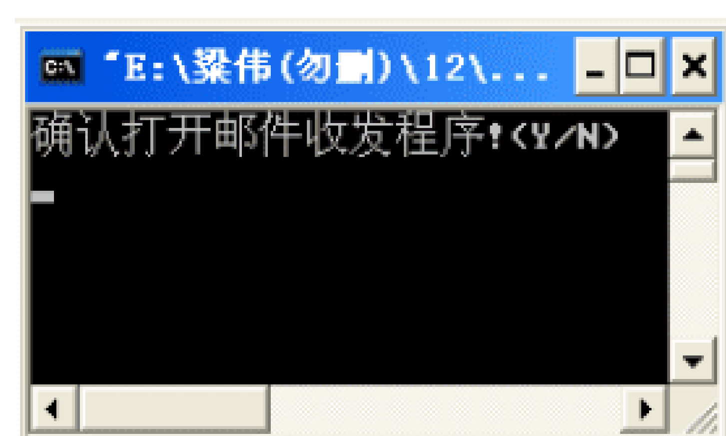


图 7.4　运行界面 1

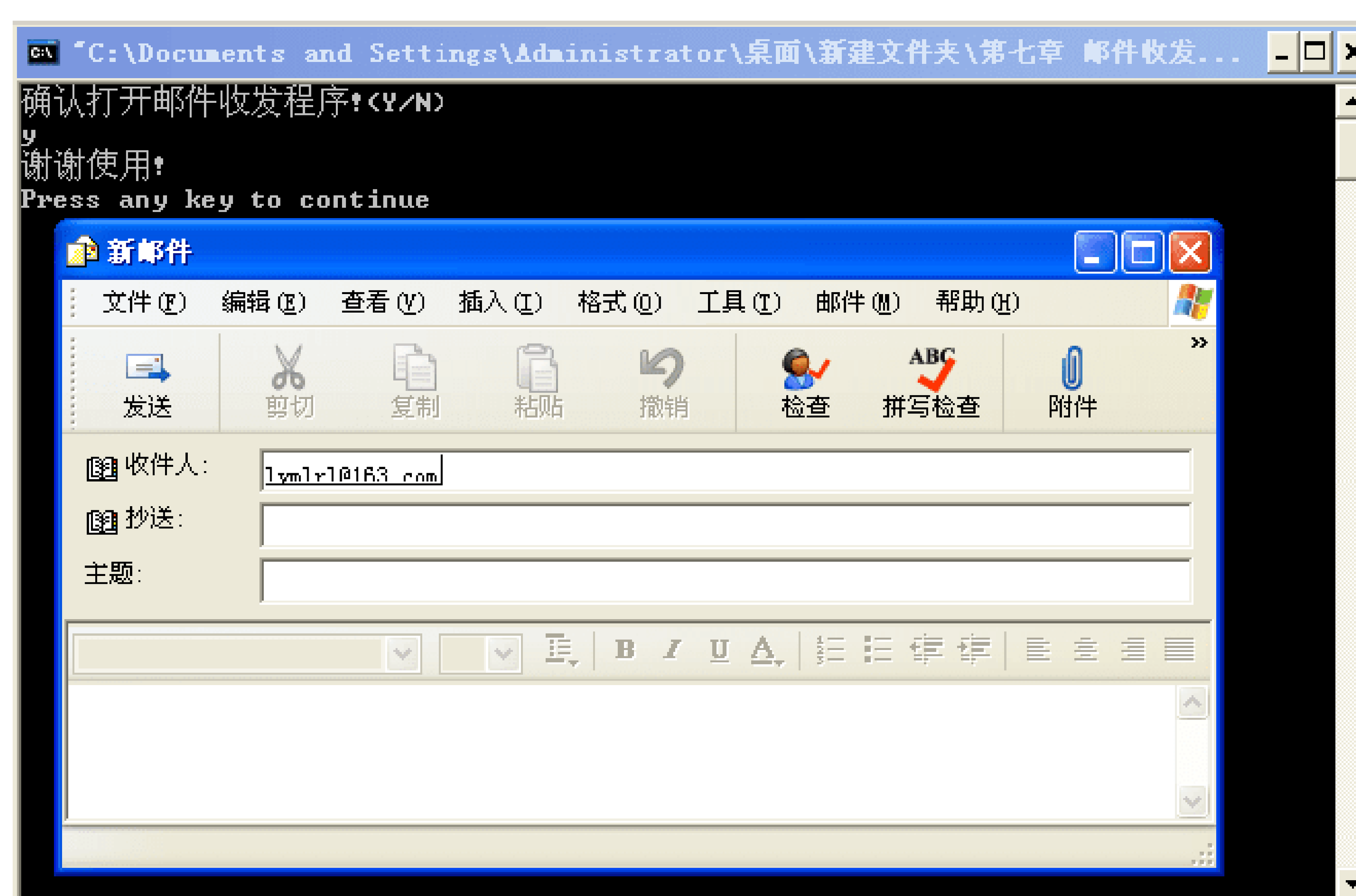


图 7.5　运行界面 2

7.2　SMTP 会话过程

SMTP 是发送邮件协议，与前面所讲的 FTP、HTTP 等协议一样被用作某种行为的规范标准。本节的主要内容就是向用户讲解邮件客户端怎么连接 SMTP 服务器以及向 SMTP 服务器发送信件等操作。

7.2.1　怎么连接服务器

在网络中传输邮件信息都是基于 TCP/IP 协议的，所以用户在 Windows 操作系统中编写邮件发送程序时可以使用 Windows 套接字来完成。一般情况下，客户端连接服务器的几个步骤如下。

（1）客户端以指定 IP 地址和端口连接服务器。

（2）服务器收到连接请求，并同意客户端连接请求。

（3）客户端和服务器互相发送数据。

（4）关闭服务器和客户端的套接字。

基于以上几个步骤，用户可以 VC 中编写程序实现邮件客户端。

1．创建套接字对象

该实例与一般网络程序一样，需要 Windows 套接字（SOCKET）的支持，所以用户应该首先初始化套接字库。代码如下：

```
BOOL CMyEMAIL::OnInitDialog()
{
...                                //省略部分代码
DWORD ss=MAKEWORD(2,0);            //指定套接字库版本
  ::WSAStartup(ss,&data);          //初始化套接字库
...                                //省略部分代码
}
```

用户初始化套接字库以后，还必须记得在程序退出之前释放该套接字库。代码如下：

```
void CMyEMAIL::OnClose()
{
    ::WSACleanup();                    //释放已经加载的套接字库
}
```

然后，用户可以调用 API 函数 socket()创建连接服务器的套接字了。代码如下：

```
BOOL CMyEMAIL::OnInitDialog()
{
...                                                    //省略部分代码
SOCKET s;                                              //定义套接字对象
sockaddr in addr;                                      //定义网络地址结构对象

addr.sin_family=AF_INET;                               //为地址结构中的成员赋值
    addr.sin port=htons(25);
    addr.sin_addr.S_un.S_addr=inet_addr(smtpip);       //设置 SMTP 服务器的地址
    s=::socket(AF_INET,SOCK_STREAM,IPPROTO_TCP);       //创建套接字
    ...                                                //省略部分代码
}
```

在代码中，用户设置了套接字将要连接服务器的端口号码是 25，该端口是标准的电子邮件传输端口，与 HTTP 协议的 80 端口一样。函数 socket()创建了基于 TCP 通信的流式套接字句柄。

2. 连接服务器

用户创建好套接字以后，可以调用 API 函数 connect()连接服务器。其原型如下：

```
int connect (
  SOCKET s,
  const struct sockaddr FAR* name,
  int namelen
);
```

该函数用于连接远程计算机，如果连接失败则返回–1，否则成功。参数及其意义如下：

- ❑ 参数 s 表示将要连接服务器的套接字句柄，该套接字是用户之前已经创建好的套接字句柄。
- ❑ 参数 name 是指向套接字地址结构体的指针变量。该套接字结构体声明如下：

```
struct sockaddr_in {
       short   sin_family;
       u short sin port;
       struct  in_addr sin_addr;
       char    sin_zero[8];
};
```

该结构体是 sockaddr 结构的扩充结构，一般被用在 Windows Socket 2 中。

- ❑ 参数 namelen 表示套接字结构对象的大小。

使用该函数在套接字 s 上连接 SMTP 服务器。例如，SMTP 服务器地址为“mail.163.com”，端口为 25。代码如下：

```
BOOL CMyEMAIL::OnInitDialog()
{
```

```
   DWORD ss=MAKEWORD(2,0);                                //指定套接字库版本
   ::WSAStartup(ss,&data);                                //初始化套接字库
   SOCKET s;                                              //定义套接字对象
hostent host;                                             //定义主机结构体对象
sockaddr_in addr;                                         //定义网络地址结构对象

addr.sin_family=AF_INET;                                  //为地址结构中的成员赋值
    addr.sin_port=htons(25);
   host=gethostbyname("mail.163.com");                    //从服务器名获取主机地址
    addr.sin_addr.S_un.S_addr=inet_addr(host-> h_addr_list[0]);
                                                          //设置 SMTP 服务器的地址
    s=::socket(AF_INET,SOCK_STREAM,IPPROTO_TCP);          //创建套接字
   ::connect(s, (sockaddr*) &addr,sizeof(addr));          //连接 SMTP 服务器
   ...                                                    //省略部分代码
}
```

通过上面的代码，用户已经向 SMTP 服务器发送了连接请求。当服务器接受客户端的连接请求以后，服务器会返回相关响应码给客户端。该响应码的前 3 位数字表示服务器端响应的结果。部分 SMTP 响应码如表 7.3 所示。

表 7.3　部分SMTP响应码

响　应　码	意　　义
220	服务器就绪
221	服务器关闭传输通道
250	客户端所请求的邮件操作完成
450	邮件地址不可用
421	服务器服务不可用，关闭传输通道
451	由于处理过程中出错，请求的操作被终止
452	服务器存储空间不足
500	SMTP 命令语法错误
501	命令参数的语法错误
502	命令暂时不可实现
503	错误的命令序列
550	客户端请求的操作不能被执行或者邮件地址不可用
552	服务器的存储不足
553	邮箱名称不合法
554	服务失败
334	发送验证用户名
235	验证账号密码失败

在该实例中，客户端如果连接服务器成功则会返回响应码 220，表示服务器服务就绪，否则返回 554。客户端接收响应码应该调用 API 函数 recv()。代码如下：

```
BOOL CMyEMAIL::OnInitDialog()
{
   ...                                                    //省略部分代码
char recvbuff[3]={0};                                     //定义接收缓冲区
sockaddr_in addr;                                         //定义网络地址结构对象
addr.sin_family=AF_INET;                                  //为地址结构中的成员赋值
    addr.sin_port=htons(25);
```

```
        host=gethostbyname("mail.163.com");        //从服务器名获取主机地址
    addr.sin addr.S un.S addr=inet addr(host-> h addr list[0]);
                                                   //设置 SMTP 服务器的地址
    s=::socket(AF INET,SOCK STREAM,IPPROTO TCP);   //创建套接字
    if(connect(s, (sockaddr*) &addr,sizeof(addr))) //连接 SMTP 服务器
    {
      recv(s,(LPSTR)recvbuff,3,0);                 //接收响应码前 3 位数字
   if(recvbuff[0]==220)
     {
       MessageBox("服务器启动服务就绪！请继续操作！");  //提示用户服务器就绪
        }
     }
}
```

本节中，向用户讲述了连接 SMTP 服务器、SMTP 响应码的具体意义以及客户端接收响应码，并且配有相关的代码实例。

7.2.2　SMTP 命令

在客户端与 SMTP 服务器之间进行数据传输时，双方都是使用 SMTP 命令进行交流。因此，SMTP 命令在 E-mail 通信中起着很重要的作用。但是，在向用户讲解 SMTP 命令之前，用户必须首先了解一下电子邮件的基本格式。

1．E-mail构造格式

在 SMTP 协议中，规定了 E-mail 信件的基本格式。该格式与第 5 章中向用户所讲述的 HTTP 基本格式一样，都包含有数据头和数据体，并且在两者之间均使用一个空白行隔开。例如，一封简单的邮件构造格式如下：

```
Data:Tue,04 Feb 2009 21:18:03+0800 //邮件头
From:lymlrl@163.com
To:lymlrl@126.com
Subject: This is a E-mail
                                   //空白行
Hello lymlrl!                      //邮件体
This is a E-mail!
```

在例子中，E-mail 的基本格式包括邮件头和邮件体。邮件头中的内容是关于该邮件的一些基本信息。例如，邮件发送的时间、发送者以及接收者等基本信息。而邮件体中是纯文本的邮件内容，并且在 SMTP 协议中，还规定在邮件头和邮件体之间需要使用一个空白行隔开。

在邮件头中，主要是由 SMTP 标准字段组成，这些字段包含邮件的基本信息。例如：

```
Data:Tue,04 Feb 2009 21:18:03+0800 //邮件头
From:lymlrl@163.com
To:lymlrl@126.com
Subject: This is a E-mail
```

以上字段所包含的信息是邮件被操作的时间，即 2009 年 2 月 4 日星期三，邮件发送者的邮件地址是 lymlrl@163.com，邮件发送目的地是 lymlrl@126.com，邮件主题是 This is a E-mail。在 SMTP 协议中，包含了很多邮件头标准字段，部分 SMTP 邮件头字段如表 7.4

所示。紧跟着邮件头的是一个空白行，用于区分邮件头和邮件体。在邮件体中，主要是邮件需要发送的信息内容。在邮件体中，不包含任何字段信息，只有文本格式的邮件内容而已。

表 7.4　SMTP邮件头字段

字　段	意　义
From	邮件创建者的邮件地址
To	邮件目的地
Sender	邮件发送者
Reply-to	邮件回复地址
Cc	邮件抄送人
In- Reply-To	邮件正被回复
Data	邮件创建的时间
Subject	邮件主题
Comments	邮件的其他说明
Keywords	邮件的关键字
Bcc	邮件的密件抄送人邮件地址
Message-ID	邮件的标识符

在表 7.4 中列出了部分 SMTP 标准字段。其中，From 表示邮件的创建者地址，该地址在一般情况下仅有一个。Sender 表示邮件的发送者，该发送者可能是转发邮件，该字段可以有多个邮件地址，地址之间使用逗号隔开。同时可以有多个地址的字段是 To。例如：

```
Data:Tue,04 Feb 2009 21:18:03+0800
From:lymlrl@163.com
Sender: lymlrl@126.com, wexs@163.com,wen@126.com,wuy@sina.com.cn
                                                      //发送者为多个地址
To:lymlrl@126.com,data@yahoo.com.cn,asj@sina.com.cn   //接收者也为多个
Subject: This is a E-mail                             //邮件主题

Hello lymlrl!                                         //邮件数据体
This is a E-mail!
```

如果邮件没有没有发送成功，则客户端应该将该邮件重新进行发送。邮件的重发必须在保证邮件内容不发生改变的情况下进行。实际上，邮件进行重发只用在原有邮件头的标题字段前加上字符串“Resent-”。例如，将上述实例中的邮件进行重发，内容如下：

```
Resent-Data:Tue,04 Feb 2009 21:18:03+0800
Resent-From:lymlrl@163.com
Resent-Sender: lymlrl@126.com, wexs@163.com,wen@126.com//发送者为多个地址
Resent-To:lymlrl@126.com,data@yahoo.com.cn            //接收者也为多个
Resent-Subject: This is a E-mail                      //邮件主题

Hello lymlrl!                                         //邮件数据体
This is a E-mail!
```

注意：在连接 SMTP 服务器成功以后，客户端在接收到服务器返回的 DATA 命令后，就可以将以上构造的邮件内容发送到 SMTP 服务器了。

2．SMTP命令

前面已经向用户介绍过客户端与 SMTP 服务器之间的交流是通过 SMTP 命令来完成的。常见的 SMTP 命令如表 7.5 所示。

表 7.5　常用SMTP命令

命　令	含　义
HELO	客户机向服务器问候
MAIL	指定邮件的发送者
RCPT	指定邮件的接收者
DATA	指示客户端或服务器端可以发送邮件内容
RSET	重新初始化会话状态
SEND	指定邮件的发送者
VRFY	验证邮件地址的有效性
NOOP	空操作
QUIT	终止会话
TURN	交换服务器与客户端

下面将参照表 7.3 中所列举的部分 SMTP 命令进行讲解。

- 命令 HELO 是在邮件客户端连接服务器成功以后，第一个发送到服务器的命令。其作用是向 SMTP 服务器问候。例如，客户端向服务器问候并表明自己的身份。内容如下：

```
HELO lymlrl<crlf>
```

其中，字符<crlf>表示结束符号。以上内容表示客户端向服务器问候并且表明自己的身份。例如，在 VC 中向服务器发送该命令，代码如下：

```
...                                          //省略部分代码
char sendmail[]={"HELO lymlrl\r\n"};         //构造命令字符串
send(s, sendmail,sizeof(sendmail),0);        //发送命令到服务器
...                                          //省略部分代码
```

- 命令 MAIL/ RCPT 分别表示指定邮件的发送和接收者。例如：

```
MAIL from:lymlrl@163.com<crlf>
RCPT To:lymlrl@126.com<crlf>
```

上述代码分别指定了邮件的发送者和接收者的邮件地址。

- 命令 DATA 是客户端发送到服务器表明客户端将要发送邮件到服务器。服务器收到该命令后会返回 SMTP 响应码到客户端，表示服务器已经准备好接收客户端的邮件数据。
- 命令 VRFY 是被用来验证某个邮件地址的有效性。例如，用户用该命令来验证自己的邮箱地址是否有效，则可以发送命令字符串“VRFY:lymlrl@163.com”到 SMTP 服务器。如果该邮箱地址是有效的地址，则服务器会返回响应码 250，表示客户端所请求的操作成功，否则返回 450，表示邮件地址无效。
- 命令 QUIT 表示终止服务器和客户端的会话。例如客户端向服务器发送该命令，代码如下：

```
...                                             //省略部分代码
char sendmail[]={"QUIT\r\n"};                   //构造命令字符串
send(s, sendmail,sizeof(sendmail),0);           //发送命令到服务器
...                                             //省略部分代码
```

当服务器接收到该命令以后，会返回响应码 220 到客户端，表示服务器已经关闭相关的数据通道。

- 命令 SEND 命令被用来指定邮件的发送者，在这里发送可以包括邮件的创建者或转发者。邮件的创建者只能为唯一，而转发者可以有多个。例如，有一封邮件是由笔者所创建，被某个用户所转发，该用户的邮件地址是 wsds@126.com，那么响应的代码如下：

```
...                                             //省略部分代码
char sendmail[]={"SEND:wsds@126.com\r\n"}; //构造命令字符串
send(s, sendmail,sizeof(sendmail),0);           //发送命令到服务器
...                                             //省略部分代码
```

服务器接收到该命令后，会返回响应码 250 表示成功。

注意： 表 7.5 中的命令在程序中被发送时必须加上换行符号“\r\n”，或者用户在构造完成整个邮件内容后，需要在邮件内容后面加上“\0”，表示数据内容发送或者接收完毕。

3. 构造邮件实例程序

下面将引导用户结合前面所讲的知识构造一封简单的邮件，并编写程序将其发送到 SMTP 服务器。首先，构造邮件格式如下：

```
Data:Tue,04 Feb 2009 21:18:03+0800
From:lymlrl@163.com
Reply-to:lymlrl@sina.com.cn                             //回复邮件地址
Sender: lymlrl@126.com, wexs@163.com,wen@126.com,wuy@sina.com.cn
To:lymlrl@126.com,data@yahoo.com.cn,asj@sina.com.cn
Subject: 新年快乐!                                      //邮件主题

祝大家新年快乐!                                         //邮件体
     程序员俱乐部 lymlrl
```

然后，在 VC 中编写代码发送该邮件，代码如下：

```
BOOL CMyEMAIL::OnInitDialog()
{
  ...                                                   //省略部分代码
char recvbuff[3]={0};                                   //定义接收缓冲区
sockaddr_in addr;                                       //定义网络地址结构对象
addr.sin_family=AF_INET;                                //为地址结构中的成员赋值
addr.sin_port=htons(25);
   host=gethostbyname("mail.163.com");                  //从服务器名获取主机地址
addr.sin addr.S un.S addr=inet addr(host-> h addr list[0]);
                                                        //设置 SMTP 服务器的地址
s=::socket(AF_INET,SOCK_STREAM,IPPROTO_TCP);            //创建套接字
   if(connect(s, (sockaddr*) &addr,sizeof(addr)))       //连接 SMTP 服务器
```

```
    {
      recv(s,(LPSTR)recvbuff,3,0);                    //接收响应码前 3 位数字
   if(recvbuff[0]==220)                               //提示用户服务器就绪
     {
CString data="Data: Tue,04 Feb 2009 21:18:03+0800\r\n";    //构造发送字符串
CString from="MAIL FROM:lymlrl@163.com\r\n";
CString reply=" Reply-to:lymlrl@sina.com.cn\r\n";
CString send=" Sender: lymlrl@126.com, wexs@163.com,wen@126.com,wuy@sina.
com.cn\r\n";
CString to=" RCPT TO:lymlrl@126.com,data@yahoo.com.cn,asj@sina.com.cn\r\n";
CString subject=" Subject: 新年快乐！";
CString text=" 祝大家新年快乐！\r\n";
text+="程序员俱乐部 lymlrl\r\n";
char sendmessage[]={data.GetBuffer(1),
                        from.GetBuffer(1),reply.GetBuffer(1),
send.GetBuffer(1),to.GetBuffer(1),
                           "DATA\r\n", subject.GetBuffer(1),
                              Text.GetBuffer(1),"QUIT\r\n","\0"
                                    }
send(s,& sendmessage,sizeof(sendmessage),0);
MessageBox("邮件发送成功！");
         }
     }
}
```

上面的代码中，首先使用套接字连接 SMTP 服务器，连接成功后，调用函数接收服务器的响应。如果该响应码为 220，则表示服务器准备就绪，客户端可以发送邮件到服务器。接着，客户端构造邮件内容。在向服务器发送邮件实际内容之前，应首先发送“DATA\r\n”通知服务器准备接收数据。邮件内容发送完毕以后还需要向服务器发送“QUIT\r\n”表示结束会话。

注意：*在构造邮件时，在每句字段完成后均加上“\r\n”，并且在整个邮件构造完毕以后需要加上字符“\0”，表示发送的数据结束。*

7.2.3　发送命令与接收响应

在客户端编程中，通常情况下客户端都是通过向 SMTP 服务器发送命令表示需要进行的操作。在表 7.5 中，已经列出了部分 SMTP 常用命令，这些命令都是在客户端连接服务器成功以后发送的。客户端发送命令以后，服务器通过向客户端发送 SMTP 响应码告知其所发送的命令是否成功或被执行。

1．邮件发送与接收

客户端与 SMTP 服务器的通信过程是通过问答形式完成的，这个过程是典型的 C/S 通信模式。下面介绍一下邮件客户端发送的命令与服务器端返回的信息。内容如下：

```
Connect sever                                  //连接服务器
220...                                         //服务器就绪
HELO lymlrl
250 hello lymlrl,Welcome!                      //服务器回应
MAIL FROM:lymlrl@163.com                       //指定邮件发送者
```

```
250(from lymlrl@163.com),sender accepted          //服务器回应
RCPT TO:lymlrl@126.com,lymlrl@sina.com.cn         //指定邮件接收者
250 (lymlrl@126.com,lymlrl@sina.com.cn),OK
DATA                                              //表示即将发送邮件内容
354 please input mail                             //服务器回应
客户端发送邮件内容
250 OK, the message is saved                      //服务器接收邮件内容完毕
QUIT                                              //客户端退出
221 BYEBYE,SEE YOU
```

通过上面的内容，用户可以看到这是发送邮件所要经历的一个典型的 C/S（客户端/服务器）通信过程，通过问答的形式将一封邮件发送到服务器。

注意：在客户端发送 DATA 命令以后，服务器会返回是否准备好接收客户端将要发送邮件的响应码，该响应码是 354，表示服务器已经准备好接收邮件。接下来，客户端可以直接将邮件发送到服务器。在上面的内容中，单数行是客户端发送的命令或操作，双数行是服务器返回的响应码。

2. 发送SMTP命令

在实例中，客户端发送命令是通过 MFC 函数 send()进行的。该函数的作用是向套接字的另一方发送指定缓冲区中的内容。函数原型如下：

```
int send(SOCKET s,const char* buff,int len,int flags);
```

该函数调用成功返回非 0 值，否则失败。部分参数意义如下：

- 参数 s 表示客户端所创建的套接字句柄。
- 参数 buff 指向缓冲区的字符指针。
- 参数 len 表示缓冲区的大小，可以使用函数 sizeof()获得。

例如，用户使用函数 send()将命令 DATA 发送到服务器，代码如下：

```
char *send;
CString str="DATA\r\n";                                 //定义命令字符串
sockaddr_in addr;                                       //定义网络地址结构对象

addr.sin family=AF INET;                                //为地址结构中的成员赋值
    addr.sin port=htons(25);
   host=gethostbyname("mail.163.com");                  //从服务器名获取主机地址
    addr.sin addr.S un.S addr=inet addr(host-> h addr list[0]);
                                                        //设置 SMTP 服务器的地址
    s=::socket(AF_INET,SOCK_STREAM,IPPROTO_TCP);        //创建套接字
   if(connect(s, (sockaddr*) &addr,sizeof(addr)))       //连接 SMTP 服务器
   {
       recv(s,(LPSTR)recvbuff,3,0);                     //接收响应码前 3 位数字
  if(recvbuff[0]==220)                                  //提示用户服务器就绪
    {
send=str.GetBuffer(1);                                  //获取字符串首地址
              send(s,&send,sizeof(send),0);             //发送字符串
       }
```

实例程序中，用户首先发送连接请求到服务器并且等待服务器的响应。如果服务器返

回的响应码是 220，则表示服务器接受客户端的请求并准备就绪。接着，客户端便可以将命令字符串通过套接字发送到服务器执行。

3. 接收邮件服务器响应

客户端接收的消息来自于服务器端返回的响应码。实现该功能的函数是 recv()，该函数原型如下：

```
int recv(SOCKET s,const char* buff,int len,int flags);
```

该函数调用成功，则返回实际接收到的字符数，否则失败。主要参数意义如下：

- 参数 s 套接字句柄。
- 参数 buff 表示接收数据的缓冲区指针，与函数 send()一样。
- 参数 len 表示将接收的数据大小。在这里将该参数设置为 3。

用户在编写客户端程序时，主要是接收服务器响应码的前 3 位数字。所以，用户在使用函数 recv()接收消息时，字符长度固定为 3 即可。代码如下：

```
...                                                    //省略部分代码

if(connect(s, (sockaddr*) &addr,sizeof(addr)))         //连接 SMTP 服务器
    {
      recv(s,(LPSTR)recvbuff,3,0);                     //接收响应码前 3 位数字
}
...                                                    //省略部分代码
```

在这里，关于客户端接收服务器响应消息的功能不再进行重复讲述，请用户复习本章前面所讲述的相关内容。

7.3　发 送 邮 件

用户通过学习前面关于邮件收发的基本原理和编程方法，对邮件收发器的制作已经熟悉。在本节中，将通过编程制作程序实例，向用户讲述在 VC 环境下编程的具体方法。通过本节实例的学习，用户可以仿照该实例的设计方法，自行编程实现邮件收发器。

7.3.1　界面设计

在任何程序中，窗口界面都是最重要的，因为程序界面直接面向用户。当用户第一次使用软件时，其窗口界面决定了用户对该软件的第一印象。所以在本实例中，程序窗口设计是基于对话框模式。用户可以在 VC 对话框资源编辑器中，通过鼠标拖动控件等方法设计一个个性化的窗口界面，然后编程实现其功能。

1. 设计界面

用户在 VC 中，可以使用工程向导创建程序实例。基本步骤如下：

（1）选择“文件”|“新建”命令，打开“新建”对话框，如图 7.6 所示。

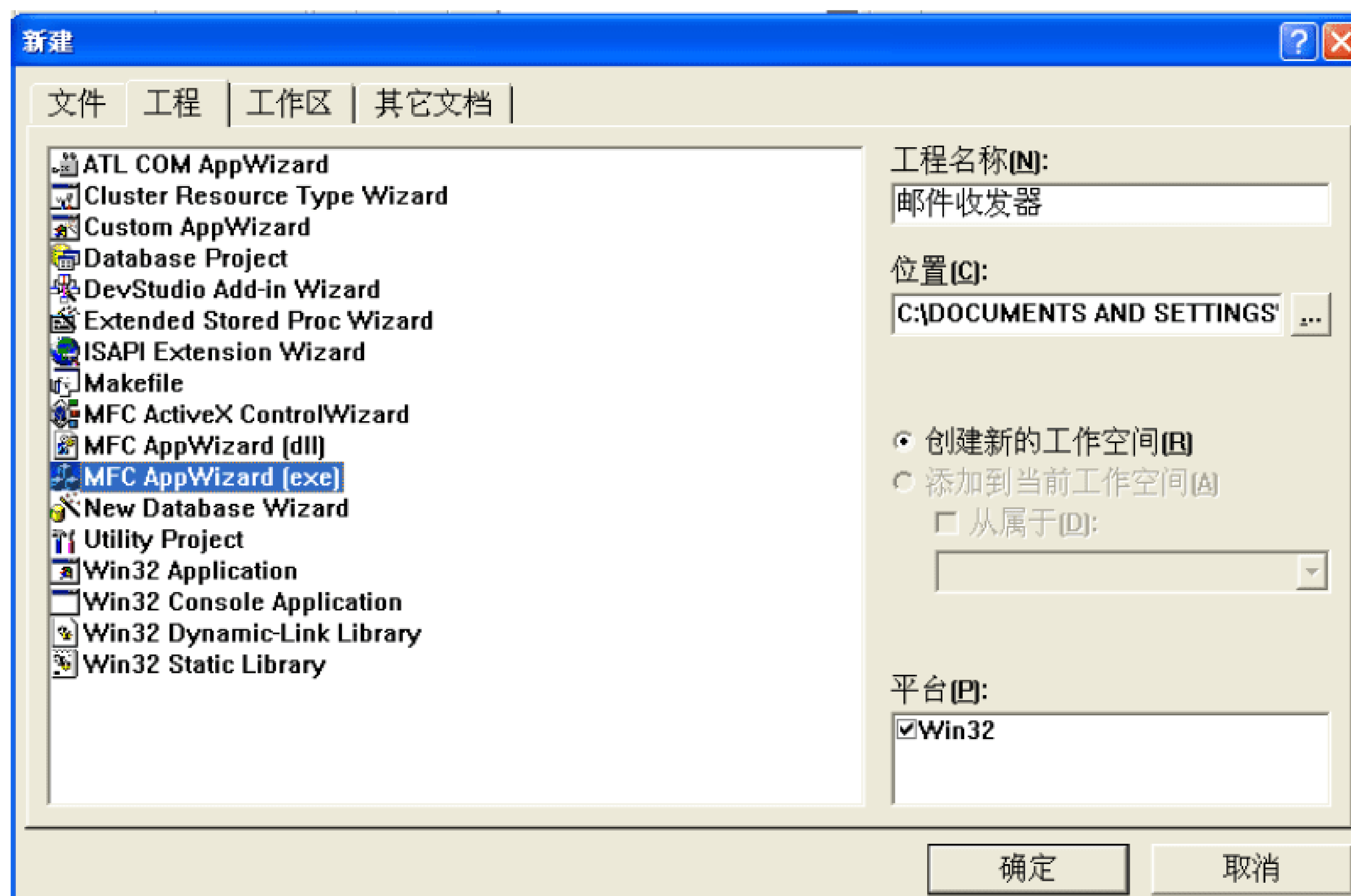


图 7.6　“新建”对话框

（2）在“新建”对话框中，用户在工程选项卡中指定创建 MFC 应用程序，在工程名称中设置为“邮件收发器”。然后单击“确定”按钮，进入下一步设置，如图 7.7 所示。

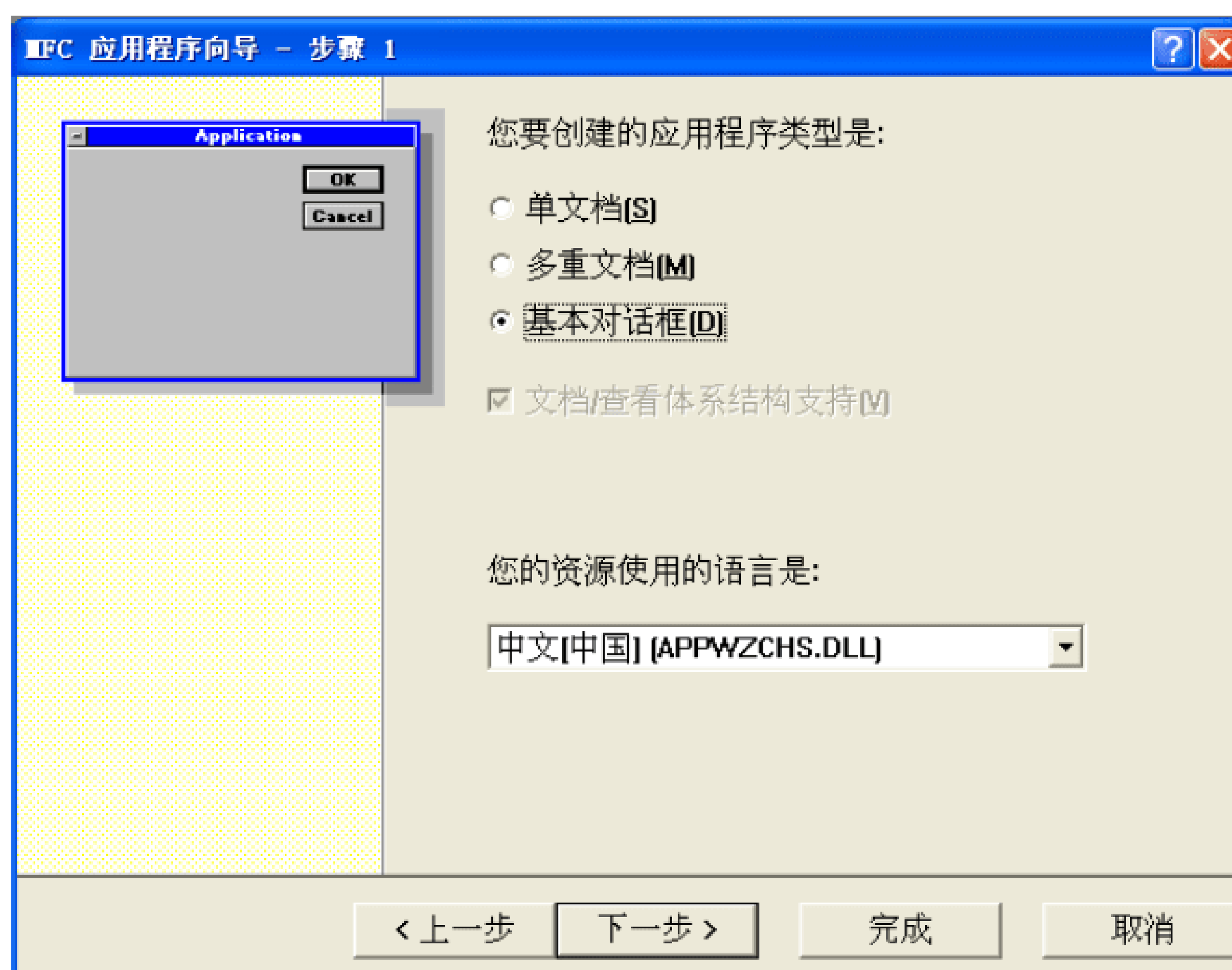


图 7.7　选择“基本对话框”单选按钮

（3）由于本实例是基于对话框的，所以在步骤 1 中，将创建的应用程序类型指定为“基本对话框”。单击“下一步”按钮，进入步骤 2，如图 7.8 所示。

（4）在步骤 2 对话框中，为项目选择支持 Windows 套接字。如果用户在这里没有选择支持套接字功能，那么，还可以在程序中加入相关代码支持套接字功能，具体添加方法将在后面的编程中介绍。单击“下一步”按钮继续设置。跳过步骤 3 进入最后一步，如图 7.9

所示。

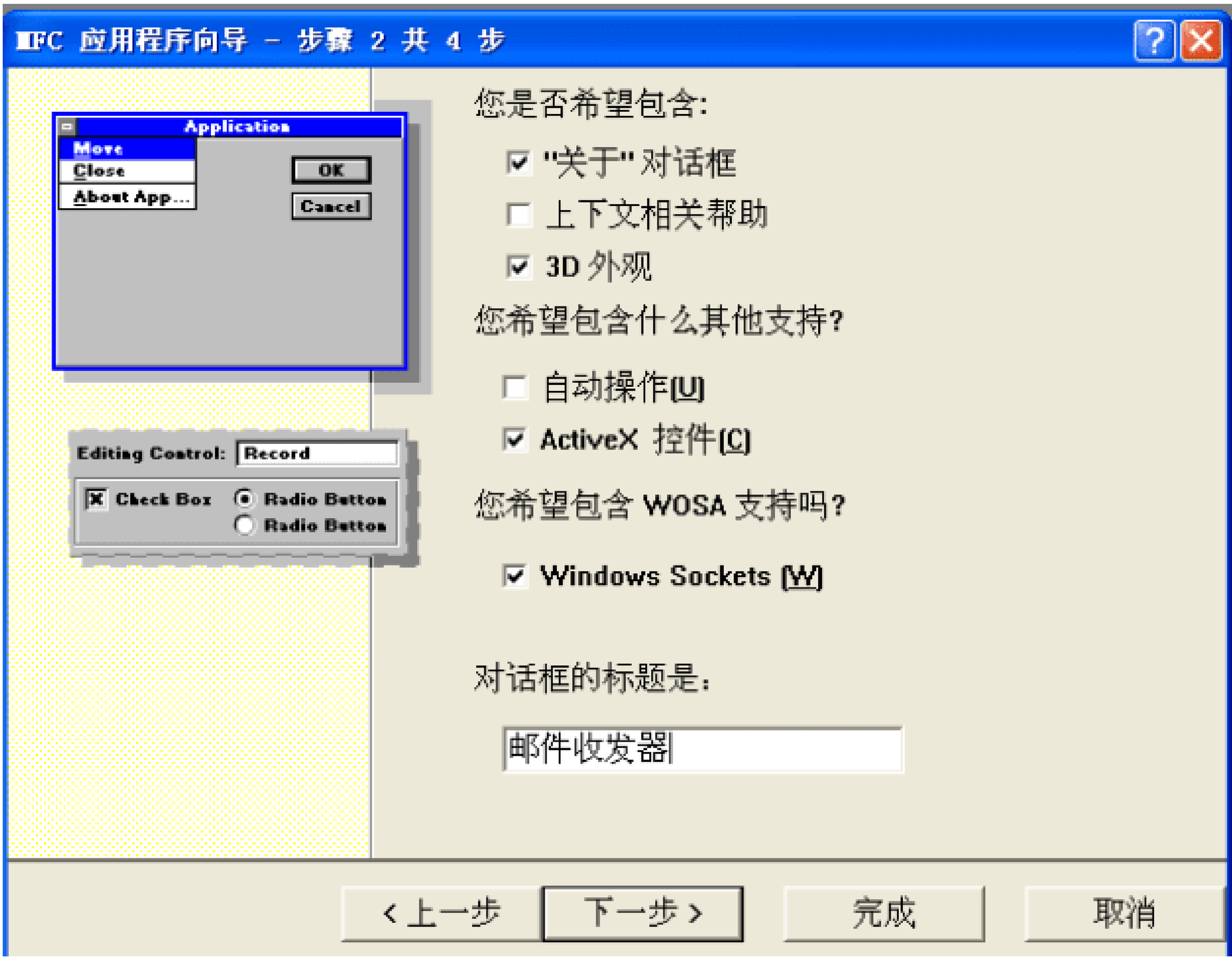


图 7.8　选择 Windows Sockets 复选框

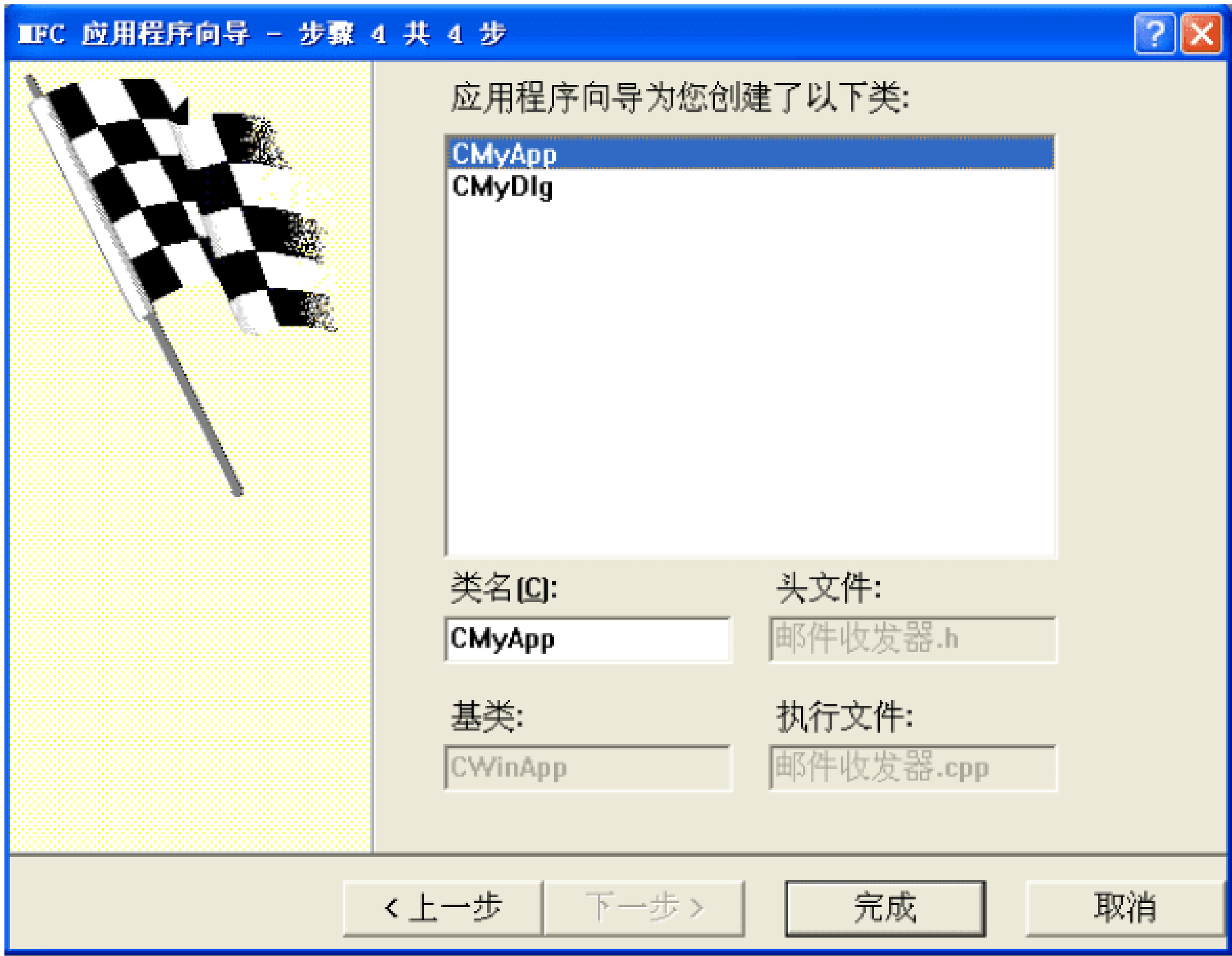


图 7.9　完成设置

（5）在最后一步中，用户可以在列表中单击该工程中的类名，查看关于该类相关文件的信息。如单击 CMyDlg 类，将会显示该类的类名、头文件名、执行文件名等信息，如图 7.10 所示。

用户仅需要单击“完成”按钮，便可以完成该工程的相关设置，并返回到 VC 主界面。

2. 添加控件

在 VC 资源管理器中编辑默认界面，如图 7.11 所示。在默认界面中添加所需控件，控件 ID 以及作用如表 7.6 所示。

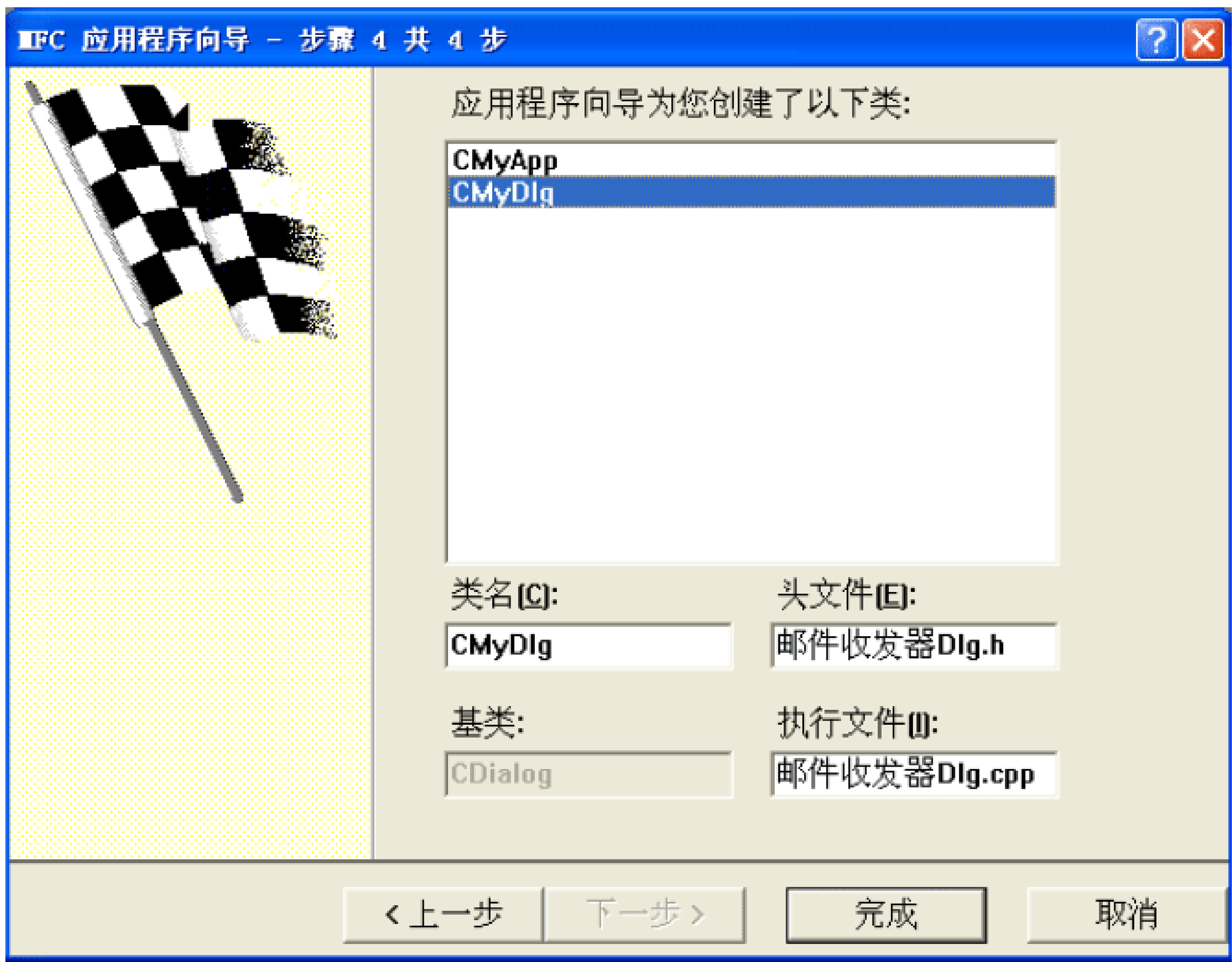


图 7.10　显示类相关信息

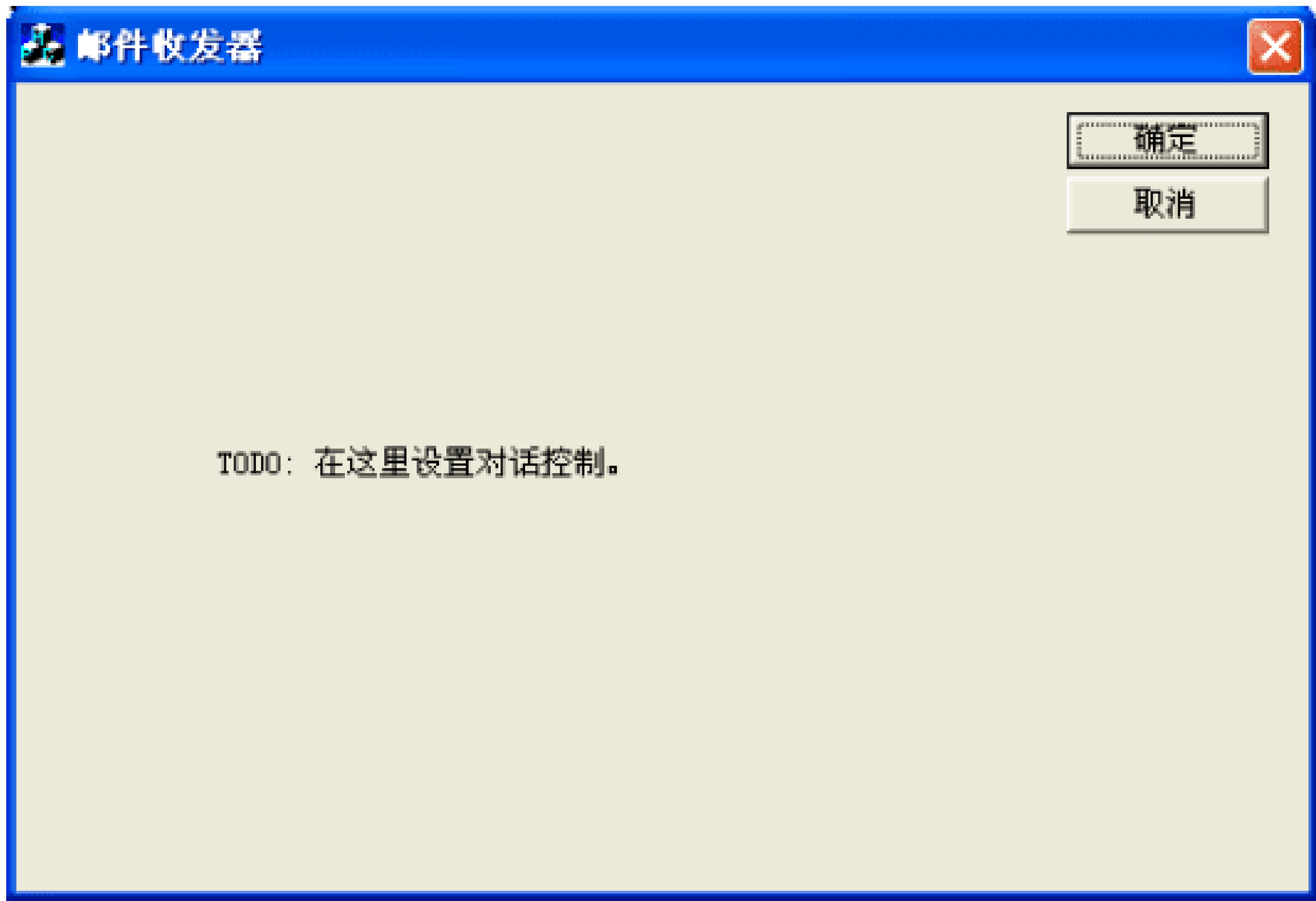


图 7.11　VC 工程默认界面

用户在默认界面中应首先删除默认界面上的按钮以及静态文本等。如果需要调整实例窗口的大小，可以直接使用鼠标拖拉或者是在后面的程序中编写相应的代码加以实现。

表 7.6　控件ID以及作用

控　件　ID	控 件 属 性	作　　用
IDC_SENDER	编辑框	邮件发送者的邮件地址
IDC_RECVER	编辑框	邮件接收者的邮件地址
IDC_SUBJECT	编辑框	邮件主题
IDC_PEIZHI	按钮	SMTP 服务器配置
IDC_SENDMAIL	按钮	发送邮件按钮
IDC_RECVMAIL	按钮	接收邮件按钮
IDC_HELP	按钮	帮助按钮
IDC_MAILTEXT	编辑框	邮件内容

用户在 VC 默认界面中添加表 7.6 中所示的控件，并调整各个控件的位置以及大小。添加并调整控件后的程序初始界面如图 7.12 所示。



图 7.12　调整控件后的程序初始界面

用户可以看到调整控件的位置与大小以后，程序的界面显得非常整齐、流畅。但是，程序窗口启动时，使用者可能并不知道如何去使用该软件，所以作为程序员应该对使用者的使用进行引导。如，应该首先进行 SMTP 相关设置，再进行邮件相关的操作。实现这类功能，用户需要在程序初始化时进行编程实现。

7.3.2　界面初始化代码

程序界面初始化时需要将软件相关信息显示给用户。所以，在本实例程序中，同样需要显示一些关于程序使用方面的帮助信息给用户参考。在本节中，将向用户讲解初始化界面的相关知识。

1．初始化界面

当实例窗口第一次启动时，程序应该在窗口中创建状态栏用于显示一些提示信息。因此，在程序初始化函数中创建状态栏并显示软件信息等。代码如下：

```
class CMyDlg : public CDialog
{
     public:
         HWND statu;                              //定义状态栏对象
...                                               //省略部分代码
}
BOOL CMyDlg::OnInitDialog()                       //程序初始化函数
{
    CDialog::OnInitDialog();                      //调用基类的初始化函数
    statu=::CreateStatusWindow(WS_CHILD|WS_VISIBLE,"欢迎使用本软件！(作者:
Liangwei)",this->m_hWnd,IDC_123);                //创建状态栏，ID 号码为 IDC_123
    this->SetWindowText("邮件收发器 v1.0");       //设置窗口标题
   // this->SetTitle("邮件收发器 v1.0");
    return TRUE;
}
```

添加代码时，首先调用函数 CreateStatusWindow()为实例程序创建一个状态栏并返回状态栏句柄，在该状态栏上显示程序的相关信息，状态栏的 ID 号码设置为 IDC_123。然后调用 SetWindowText()或者 SetTitle()函数将实例窗口的标题设置为“邮件收发器 v1.0”，标题包含了程序版本号。最后在文件“邮件收发器.rc”中，定义状态栏的 ID。代码如下：

```
#define  IDC_123  1008
```

另外，为了让用户首先查看该程序的使用方法，应当在程序初始化时，除了“使用前查看”按钮处于可用状态之外，其他按钮应该全部禁用。在程序窗口初始化函数 CMyDlg::OnInitDialog()中添加代码如下：

```
BOOL CMyDlg::OnInitDialog()                                    //程序初始化函数
{
...                                                            //省略部分代码
GetDlgItem(IDC_SENDER)->EnableWindow(false);                   //设置各个控件状态
GetDlgItem(IDC_RECVER)->EnableWindow(false);
GetDlgItem(IDC_SUBJECT)->EnableWindow(false);
GetDlgItem(IDC_PEIZHI)->EnableWindow(false);
GetDlgItem(IDC_SENDMAIL)->EnableWindow(false);
GetDlgItem(IDC_RECVMAIL)->EnableWindow(false);
GetDlgItem(IDC_MAILTEXT)->EnableWindow(false);
GetDlgItem(IDC_SENDER)->SetWindowText("请用户首先查看'使用前须知'! ");
                                                               //设置提示信息
...                                                            //省略部分代码
}
```

注意：在程序中使用 SetWindowText()函数设置控件等标题时，如果需要将某一段文字括上，那么不能再使用双引号，只能使用单引号。因为在 VC 环境下，在双引号中再使用该符号，编译器会报错。

运行以上代码，会使窗口内（除了“使用前须知”按钮）的全部控件都处于禁用状态，如图 7.13 所示。

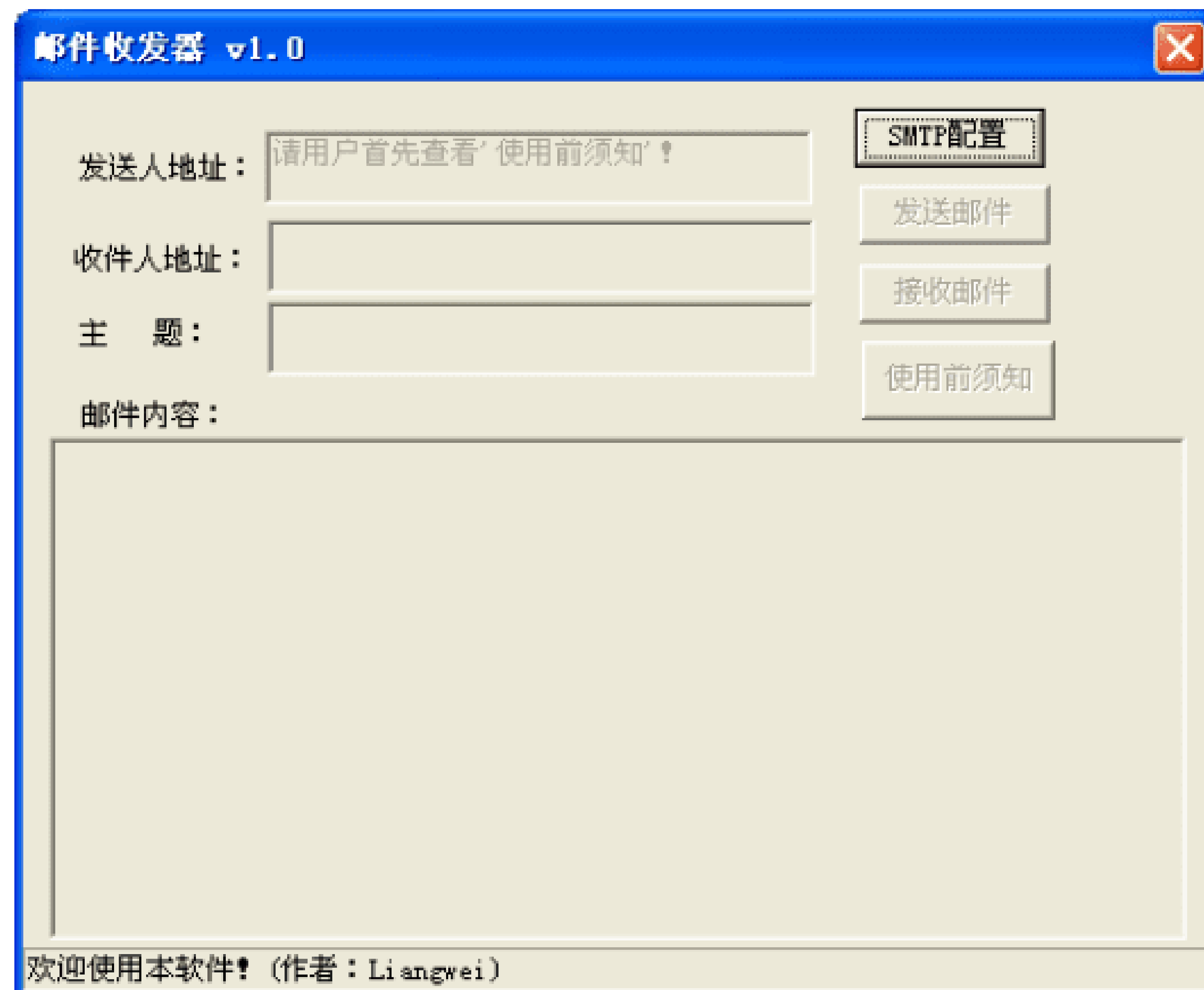


图 7.13　控件处于禁用状态

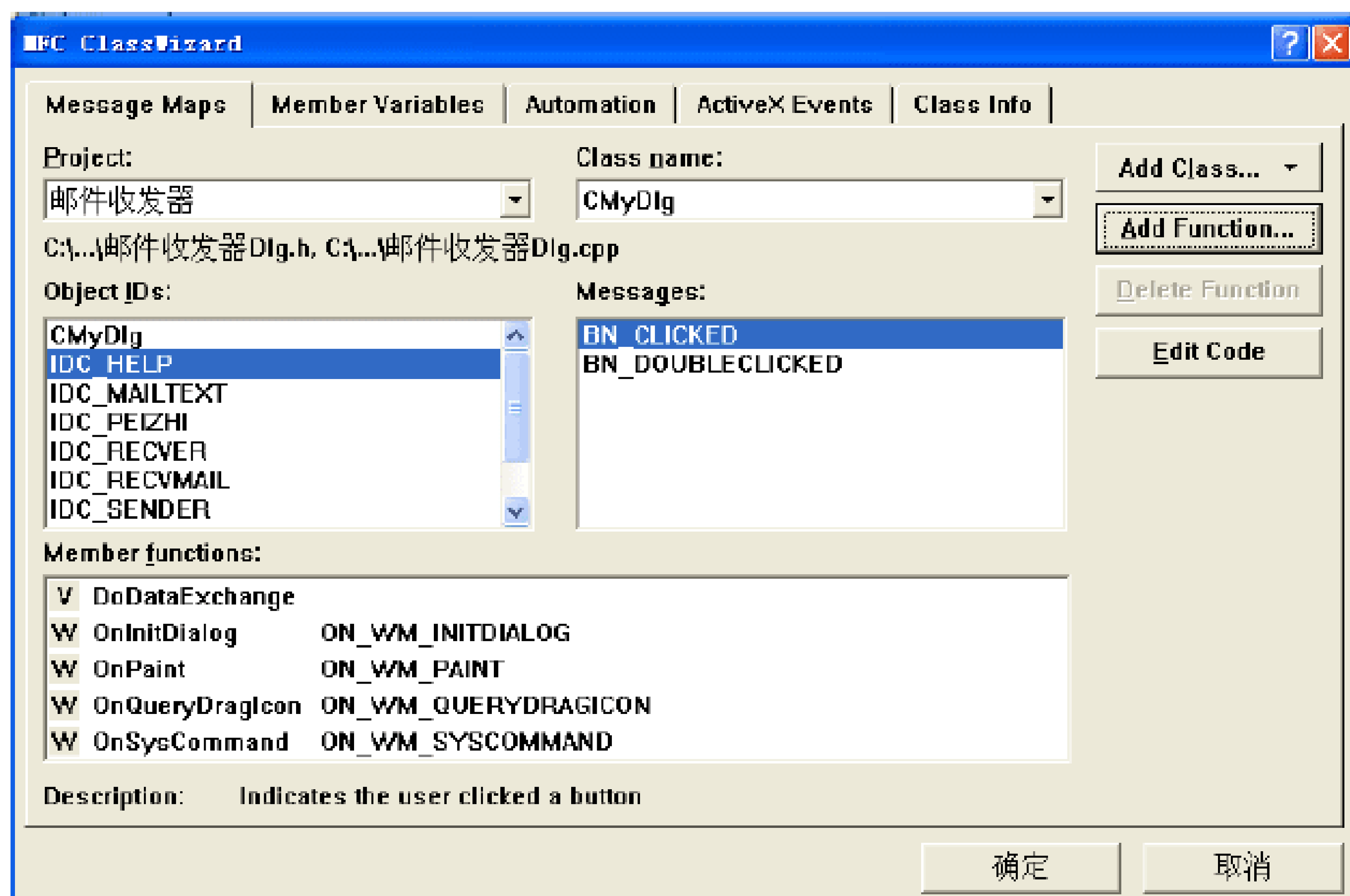
用户从图 7.13 中可以看到，界面中的控件均处于禁用状态。如果用户需要继续使用该

程序则必须按照编程者的引导步骤进行使用。在这里，主要为了方便向用户讲解程序运行的原理才设置了引导步骤。

2．添加消息响应函数

为了实现引导步骤，用户需要为“使用前须知”按钮添加消息响应函数，在该函数中实现引导使用者的功能。添加响应函数的方法是在 VC 主界面中，使用键盘上的 Ctrl+W 组合键，弹出 MFC ClassWizard 对话框，如图 7.14 所示。

用户需要在 MFC ClassWizard 对话框中，首先在 ID 列表中选择 IDC_HELP，在 Messages 列表中选择 BN_CLICKED，然后单击 Add Function 按钮弹出 Add Member Function 对话框，如图 7.15 所示。



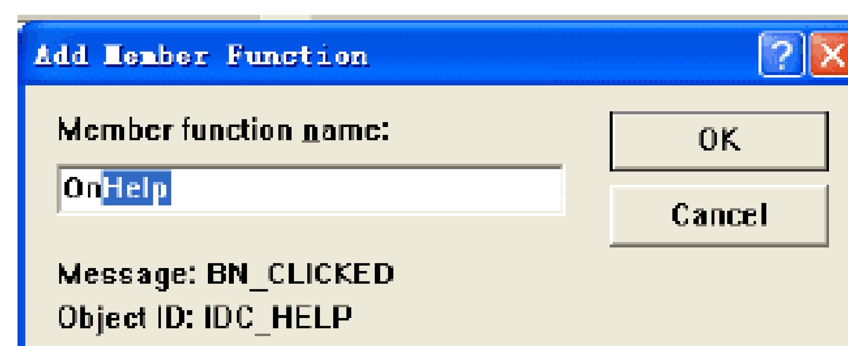


图 7.14　MFC ClassWizard 对话框　　　　图 7.15　Add Member Function 对话框

用户在图 7.15 所示对话框中可以看到 IDC_HELP 按钮的响应消息是单击消息，函数名是 OnHelp()，单击 OK 按钮返回到代码编辑器中编写代码。在该实例中，是将软件相关的使用步骤暂时显示在邮件内容编辑框中的。代码如下：

```
void CMyDlg::OnHelp()
{
    CString str;                            //构造字符串提示用户
          str+="本程序的使用方法：";
      str+="\r\n";
          str+="第一步：设置 SMTP 服务器，包括服务器地址、端口号码";
str+="\r\n";
          str+="第二步：设置发件人地址、收件人地址、邮件主题、邮件内容，";
      str+="其中，发件人地址与邮件内容可以为空，其余均不能为空。";
      str+="\r\n";
          str+="注意：如果需要将邮件发送到多人，请在收件人地址内使用逗号将地址区分开
         即可";
          str+="\r\n";                       //添加回车换行符号
          str+="作者：liangwei,QQ:393817181";//显示作者联系方式，便于学习交流
       MessageBox(str);                     //显示帮助信息
...                                         //省略部分代码
}
```

在 VC 编译器中运行程序，然后单击“使用前须知”按钮，会弹出帮助信息，如图 7.16

所示。

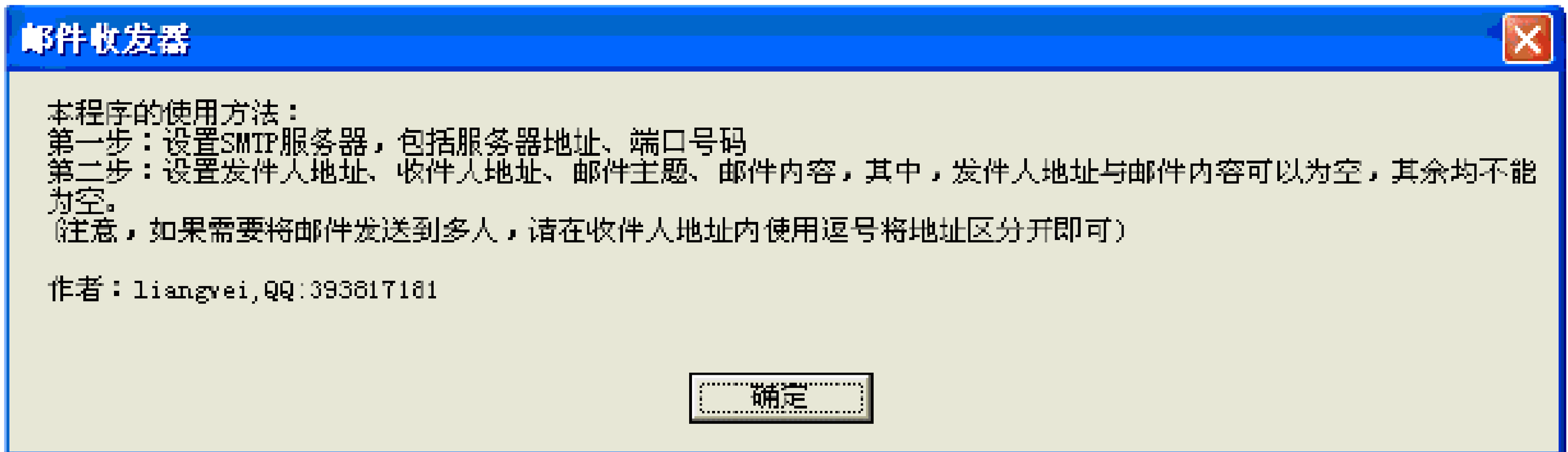


图 7.16　显示帮助信息

用户查看完帮助信息以后，程序应该使“SMTP 设置”按钮处于可用状态便于用户设置，而帮助按钮设置为禁用状态，并且将该按钮的标题设置为“功能待用”。目的是将该按钮作为扩展功能之用。实现这些功能，需要在函数 OnHelp()中修改代码如下：

```
void CMyDlg::OnHelp()
{
    CString str;                                    //构造字符串提示用户
    ...                                             //省略部分代码
      if(MessageBox(str)==IDOK)                     //显示帮助信息
       {
          GetDlgItem(IDC_HELP)->SetWindowText("功能待用");  //更改按钮标题
          GetDlgItem(IDC_HELP)->EnableWindow(false);
                                                    //禁用按钮 IDC_HELP
          GetDlgItem(IDC_PEIZHI)-> EnableWindow(true);
                                                    //设置“配置”按钮可用
}
```

当用户单击“确定”按钮以后，界面中的部分控件状态以及其标题都会发生改变，如图 7.17 所示。



图 7.17　部分控件发生改变

7.3.3　添加服务器设置对话框

为了方便用户设置 SMTP 服务器相关信息，所以在工程中需要添加一个对话框完成服

务器设置功能。在本节中，将向用户讲解添加服务器设置对话框以及在实例程序中怎样使用该对话框。

当 SMTP 配置按钮可用时，用户可以进入引导步骤的第二步设置 SMTP 服务器相关信息。首先，在 VC 中添加设置服务器对话框，ID 设置为 IDD_DIALOG1。在 VC 资源管理器中，插入对话框资源，如图 7.18 所示。对话框插入以后，添加并调整控件的位置以及大小。界面效果如图 7.19 所示。

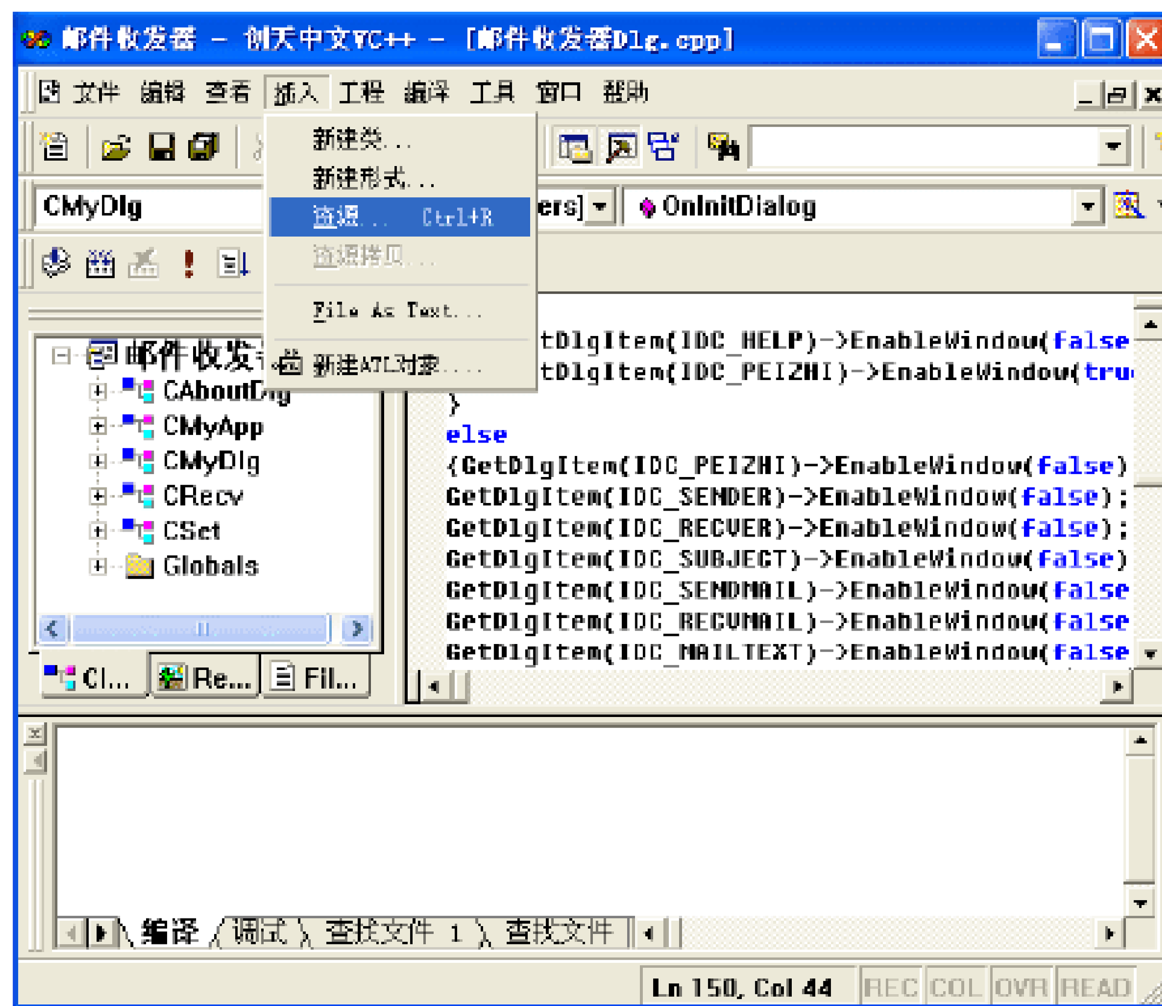


图 7.18　插入对话框

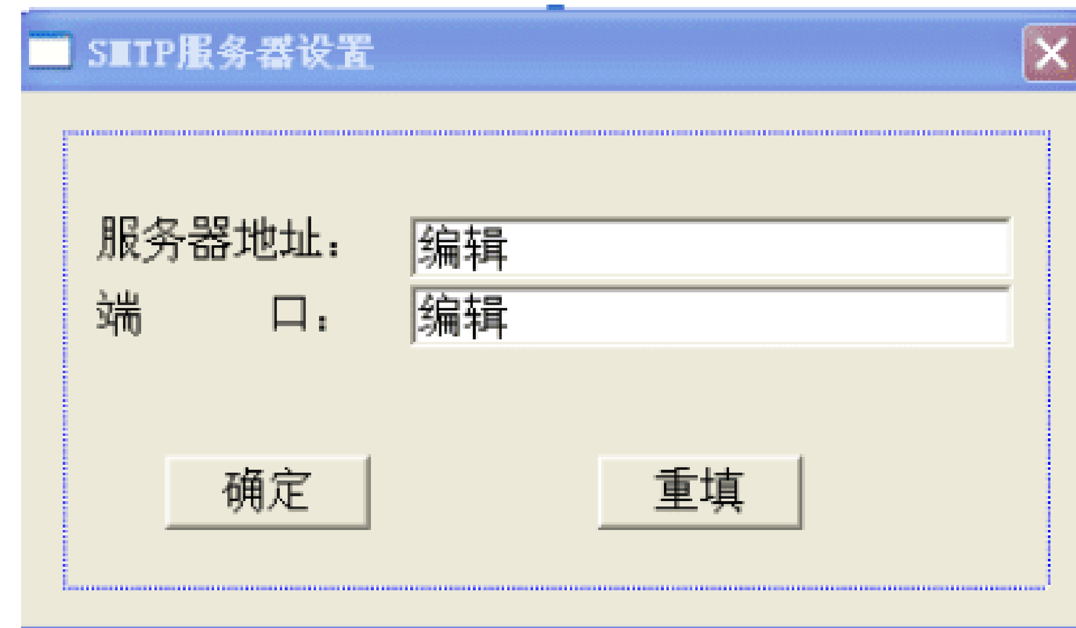


图 7.19　界面效果

然后，使用 MFC 向导为刚创建的对话框添加类，按下 Ctrl+W 组合键，弹出 New Class 对话框，直接单击 OK 按钮，弹出 New Class 对话框，如图 7.20 所示。

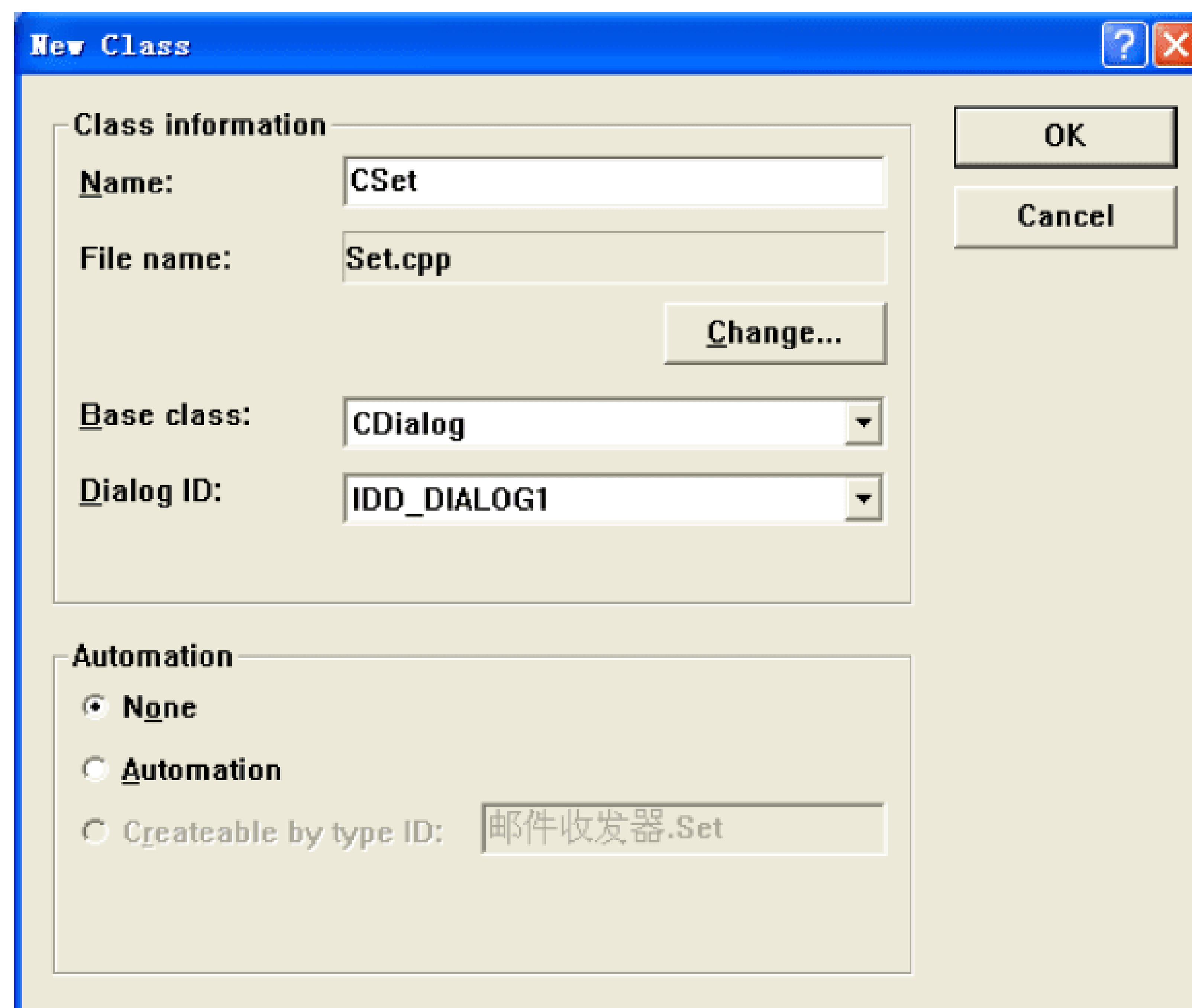


图 7.20　New Class 对话框

在对话框中，设置新类名为 CSet，基类为 CDialog。单击 OK 按钮，返回 MFC 向导对话框。在消息映射属性页中分别为确定按钮和重填按钮添加消息响应函数，如图 7.21 所示。

到这一步，服务器设置对话框已经添加完成。

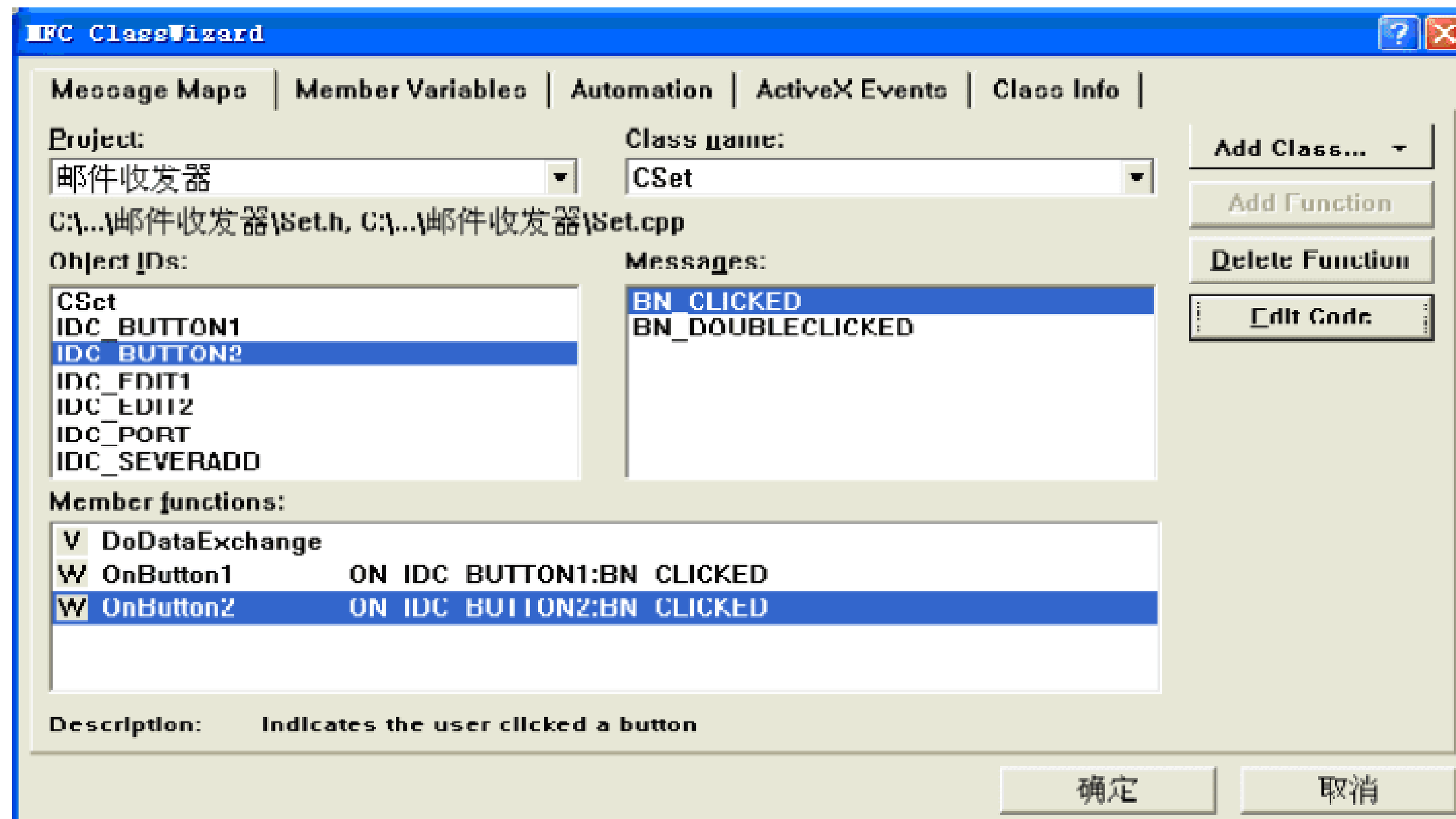


图 7.21　添加消息映射

接下来，应在 CSet 类的头文件 Set.h 中，定义两个变量用来表示服务器地址与端口号码。代码如下：

```
class CSet : public CDialog
{
  public:
      CString m_severadd;        //定义服务器地址变量
      int m_port;                //第一端口变量
...                              //省略部分代码
}
```

注意：定义变量时，为了方便在该类外部访问，应将其访问权限设置为公共。

为 CSet 类添加初始化函数 CSet::OnInitDialog()，在该函数中为服务器地址以及端口号设置默认值，并且在确定按钮和重填按钮的消息响应函数中添加代码。代码如下：

```
BOOL CSet::OnInitDialog()                                //CSet 初始化函数
{
    CDialog::OnInitDialog();
    m_severadd="mail.163.com";                           //初始化变量
        GetDlgItem(IDC EDIT1)->SetWindowText(m severadd);
                                                         //设置服务器地址
    GetDlgItem(IDC_EDIT2)->SetWindowText("25");          //设置端口号
    return TRUE;
}
void CSet::OnOK()                                        //确定按钮函数
{
    CString str;                                         //临时变量
    GetDlgItem(IDC_EDIT1)->GetWindowText(m_severadd);    //获得服务器地址
    GetDlgItem(IDC_EDIT2)->GetWindowText(str);           //获取端口号码
m_port=atoi(str.GetBuffer(1));                           //转换端口为数字型
::SendMessage(this->m_hWnd,WM_CLOSE,0,0);                //关闭设置窗口
}
void CSet::OnReset()                                     //重填按钮函数
{
```

```
    GetDlgItem(IDC_EDIT1)->SetWindowText("");          //设置两个编辑框为空
    GetDlgItem(IDC_EDIT2)->SetWindowText("");
}
```

通过上面的步骤，用户已经将服务器设置对话框成功添加到本实例的工程中了。并且为其添加了相应的类，实现了按钮的消息响应等操作，在确定按钮的函数中必须在用户设置完成以后关闭设置对话框，通过想该对话框发送 WM_CLOSE 消息。

注意： 在 SMTP 服务器时，其默认端口为 25。笔者建议使用该默认端口号，不需要进行修改。

7.3.4　使用服务器设置对话框

用户在主对话框类中使用该对话框类，必须在头文件“邮件收发器 Dlg.h”中包含新类的头文件。并且在 CMyDlg 类中定义新类对象。代码如下：

```
#include "Set.h"                       //在头文件“邮件收发器 Dlg.h”中添加
class CMyDlg : public CDialog          //在 CMyDlg 类中定义新类对象
{
  ...                                  //省略部分代码
protected:
CSet set;                              //定义 CSet 对象
}
```

在 CMyDlg 类中 CSet 类对象时将访问权限设置为保护类型。接下来，用户需要为 SMTP 设置按钮添加消息响应函数，函数名为 OnPeizhi()。在该函数中，使用 CSet 对象调用服务器设置对话框，代码如下：

```
void CMyDlg::OnPeizhi()
{
    set. DoModal();                    //调用设置对话框
    ...                                //省略部分代码
}
```

如果用户将 SMTP 服务器的相关信息设置完成以后，实例界面中的所有按钮以及编辑框等控件应该全部可用。所以，还应该在配置函数中添加代码。代码如下：

```
void CMyDlg::OnPeizhi()
{
  set. DoModal();                                      //调用设置对话框
  if(set.m_port>0 && set.m_port<100)                   //判断端口号范围
  {
    if(set.m_severadd!="")                             //判断 IP 地址不为空
{
GetDlgItem(IDC_SENDER)->EnableWindow(true);            //设置各个控件状态
GetDlgItem(IDC_RECVER)->EnableWindow(true);
GetDlgItem(IDC_SUBJECT)->EnableWindow(true);
GetDlgItem(IDC_SENDMAIL)->EnableWindow(true);
GetDlgItem(IDC_RECVMAIL)->EnableWindow(true);
GetDlgItem(IDC_MAILTEXT)->EnableWindow(true);
GetDlgItem(IDC_SENDER)->SetWindowText("");
::SendMessage(statu,SB_SETTEXT,0,(long)"SMTP 服务器信息设置成功并已经连接服务
```

```
器！");
        }
MessageBox("服务器地址不能为空");
else
      {
            MessageBox("端口范围(0～100)");
      }
}}
```

运行代码，如果用户输入服务器地址为空、端口号为空或者超出规定范围，则程序会提示用户出现错误，应该重新进行设置，如图 7.22 所示。否则，服务器设置成功，如图 7.23 所示。

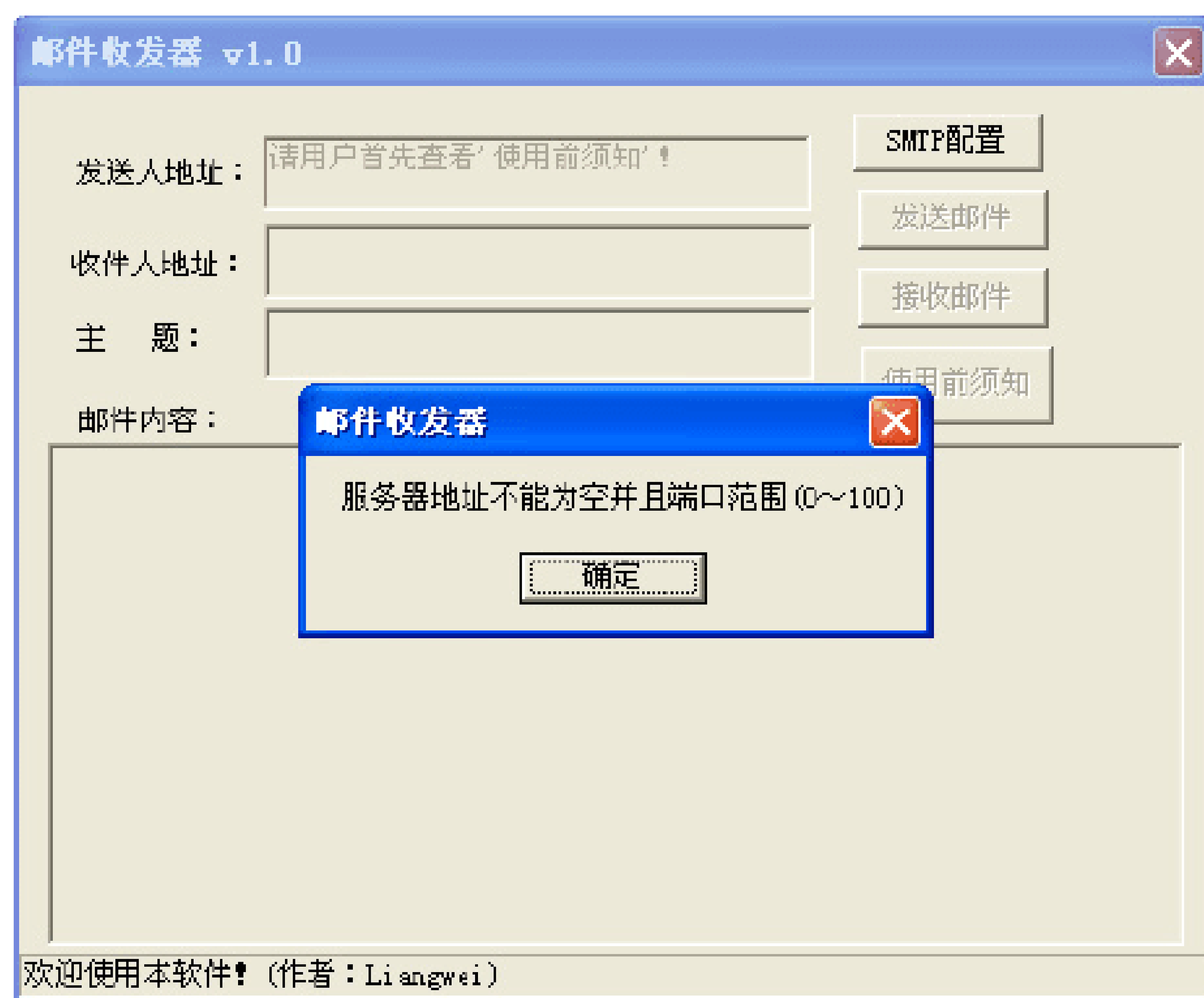


图 7.22　提示用户重新设置



图 7.23　服务器信息设置成功

SMTP 服务器信息配置成功以后，用户可以发送邮件数据到指定的 SMTP 服务器上。关于发送与接收邮件的功能将分别在下面的内容中讲解。如果用户对邮件收发器的基本工作原理和编程方法还不熟悉，请再复习本章前几小节的内容。

7.3.5　记录程序配置信息

用户在程序中，还需要实现程序的相关配置信息存取，以便用户知晓软件使用过程中的一些信息。在本实例中，笔者打算使用文件存取来实现该功能，当然也可以使用程序读取 INI 文件或者是注册表实现。

由于在本程序中，只有当用户第一次使用软件时才会被要求查看使用须知，而在后面的使用中都不被要求查看，所以在这里需要创建文件保存用户是否已经查看。如果用户已经查看，则在程序启动时通过读取查看状态决定界面的初始化。首先，在程序主对话框类中定义 CFile 对象。代码如下：

```
class CMyDlg : public CDialog
{
...                                       //省略部分代码
public:
    CFile file;                           //定义文件对象为该类属性
protected:
    HICON m_hIcon;
```

```
    CSet set;                               //设置对话框对象
...                                         //省略部分代码
}
```

用户在类中定义的文件对象 file 在该类中处于全局作用。然后，在“使用前须知”按钮的响应函数 CMyDlg::OnHelp()中添加代码如下：

```
void CMyDlg::OnHelp()
{
    ...                                               //省略先前的代码
   if(MessageBox(str)==IDOK)
{
GetDlgItem(IDC_HELP)->SetWindowText("功能待用");      //将按钮留为待用
GetDlgItem(IDC_HELP)->EnableWindow(false);            //禁用按钮
GetDlgItem(IDC_PEIZHI)-> EnableWindow(true);          //配置按钮可用
CFile file1("状态配置文件.lw",CFile::modeReadWrite);
                                        //创建文件，并将文件属性指定为可读可写
      char d=' Y';                                    //定义查看状态标志
      file1.Write(&d,sizeof(d));                      //写入该状态标志
      file1.Close();                                  //关闭文件
}
}
```

在代码中，如果用户查看使用须知并且返回，则调用 CFile 类创建文件，文件名称为“状态配置文件.lw”，文件属性为“CFile::modeReadWrite”。将查看状态标志 Y 写入配置文件中待程序启动时读取以便判断用户是否已经查看使用须知，然后关闭文件。

接下来，用户应该在程序初始化函数 CMyDlg::OnInitDialog()中，实现文件的读操作，判断查看状态以及初始化界面等。代码如下：

```
BOOL CMyDlg::OnInitDialog()                           //程序初始化函数
{
CDialog::OnInitDialog();
file.Open("状态配置文件.lw",CFile::modeReadWrite);    //以读写方式打开配置文件
char d;                                               //定义状态标志字符
file.Read(&d,1);                                      //读取状态标志
file.Close();                                         //关闭文件
if(d==' Y')                                           //判断状态标志
{
    GetDlgItem(IDC_HELP)->EnableWindow(false);        //初始化程序界面
    GetDlgItem(IDC_PEIZHI)->EnableWindow(true);
}
else
{
GetDlgItem(IDC_PEIZHI)->EnableWindow(false);
}
GetDlgItem(IDC_SENDER)->EnableWindow(false);          //设置各个控件状态
GetDlgItem(IDC_RECVER)->EnableWindow(false);
GetDlgItem(IDC_SUBJECT)->EnableWindow(false);
GetDlgItem(IDC_SENDMAIL)->EnableWindow(false);
GetDlgItem(IDC_RECVMAIL)->EnableWindow(false);
GetDlgItem(IDC_MAILTEXT)->EnableWindow(false);
GetDlgItem(IDC_SENDER)->SetWindowText("请用户首先查看“使用前须知”！");
}
```

在程序初始化函数中添加以上代码，并且运行。程序启动界面如图 7.24 所示。

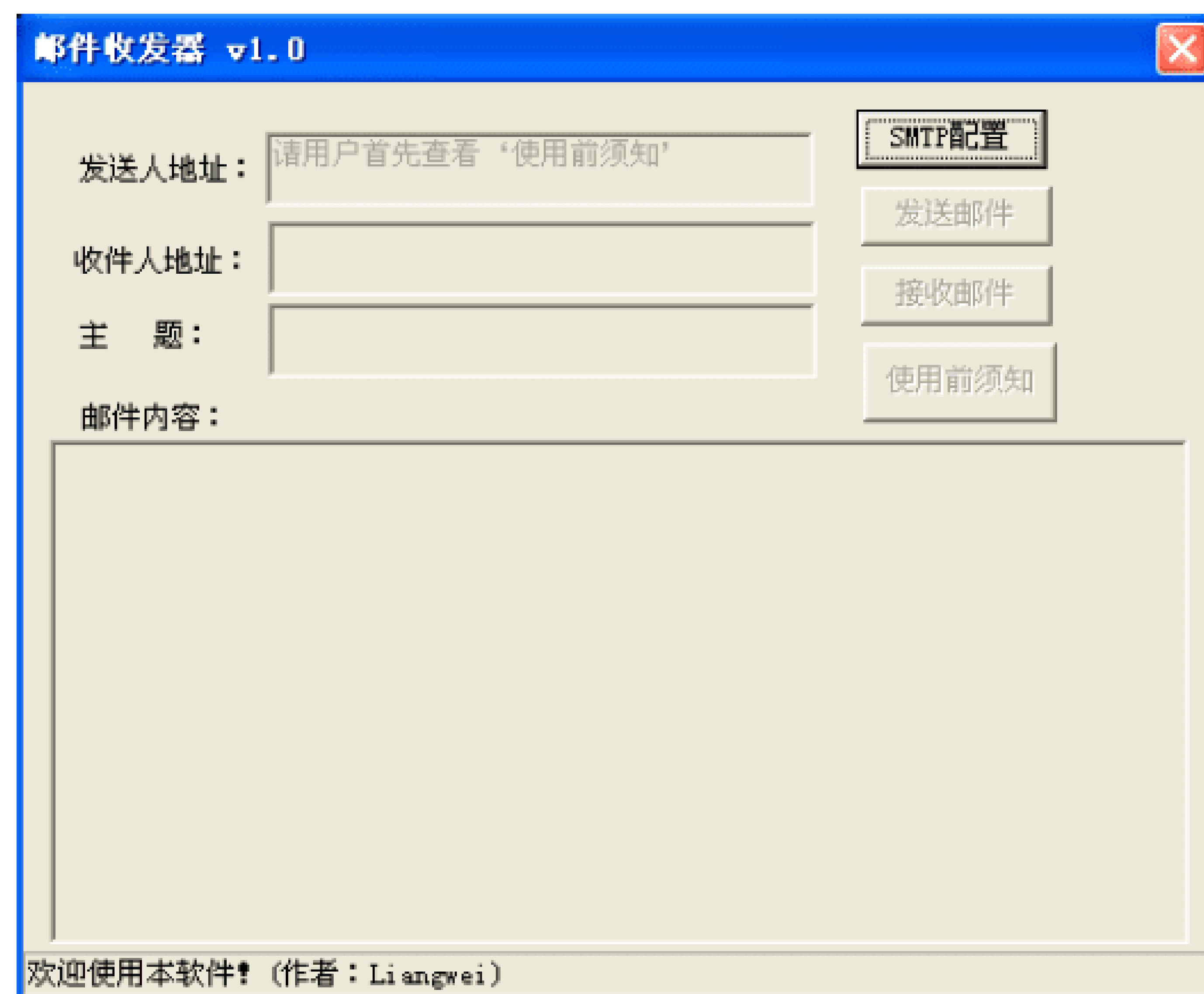


图 7.24　设置查看状态标志后的启动界面

通过上面的代码已经实现了用户仅在第一次使用软件时，需要查看程序使用须知，在以后的使用中并不需要再进行查看该须知的功能。

在本节中，用户应该了解了怎样设置程序的引导步骤，并且在该步骤中如何引导使用者正确地使用软件。在 VC 环境中，用户应该知道怎样使用 MFC 向导添加所需资源，如类和对话框等。文件操作对任何一个程序而言，都是非常重要的，通过实例代码，用户了解了如何利用文件控制程序界面的初始化操作等。

7.3.6　设置并连接服务器

在 7.3.5 节中，用户已经知道了添加服务器设置对话框的方法。那么，用户设置服务器信息完成后，应该立刻连接到该服务器。在实现该功能之前，需要在对话框类中定义套接字等变量。代码如下：

```
class CMyDlg : public CDialog
{
public:
SOCKET s;                                        //定义套接字
sockaddr_in addr;                                //定义网络地址结构对象
hostent *host;                                   //定义主机信息结构变量
}
```

用户在程序初始化函数中定义了 3 个变量，分别是套接字句柄、网络地址结构对象和主机信息结构。功能的实现应该在函数 CMyDlg::OnPeizhi()中添加代码如下：

```
void CMyDlg::OnPeizhi()                          //配置按钮响应函数
{
    // TODO: Add your control notification handler code here
    set.DoModal();                               //调用模式对话框
    if(set.m_port>0 && set.m_port<100)           //判断端口范围
  {
    if(set.m_severadd!="")                       //服务器地址不能为空
    {
addr.sin_family=AF_INET;                         //为地址结构中的成员赋值
```

```
addr.sin_port=htons(set.m_port);
//host=::gethostbyname(set.m_severadd.GetBuffer(1));   //获取主机地址
addr.sin_addr.S_un.S_addr=inet_addr(set.m_severadd.GetBuffer(1));
                                                       //转换 IP 地址
s=::socket(AF_INET,SOCK_STREAM,IPPROTO_TCP);           //创建套接字
if(connect(s, (sockaddr*) &addr,sizeof(addr)))
{
::SendMessage(statu,SB_SETTEXT,0,(long)"SMTP 服务器信息设置成功并已经连接服务
器！");
GetDlgItem(IDC_SENDER)->EnableWindow(true);            //设置各个控件状态
GetDlgItem(IDC_RECVER)->EnableWindow(true);
GetDlgItem(IDC_SUBJECT)->EnableWindow(true);
GetDlgItem(IDC_SENDMAIL)->EnableWindow(true);
GetDlgItem(IDC_RECVMAIL)->EnableWindow(true);
GetDlgItem(IDC_MAILTEXT)->EnableWindow(true);
GetDlgItem(IDC_SENDER)->SetWindowText("");
  }
else
{
    MessageBox("请检查网络连接或重新设置服务器信息！");
}
    }
    else
{
MessageBox("服务器地址不能为空");
}
    }
else
    {
        MessageBox("端口范围(0～100)");
    }
}
```

在程序中，函数 inet_addr()是将主机字节顺序的 IP 地址转换为网络字节顺序。因此，用户填写服务器地址时应使用服务器的 IP 地址而不能是服务器域名地址，如 mail.163.com。用户从网络域名地址得到 IP 地址，可以使用 ping 命令获取。其方法是选择“开始”|“运行”命令，打开“运行”对话框，输入 cmd 命令调用命令行窗口，如图 7.25 所示。

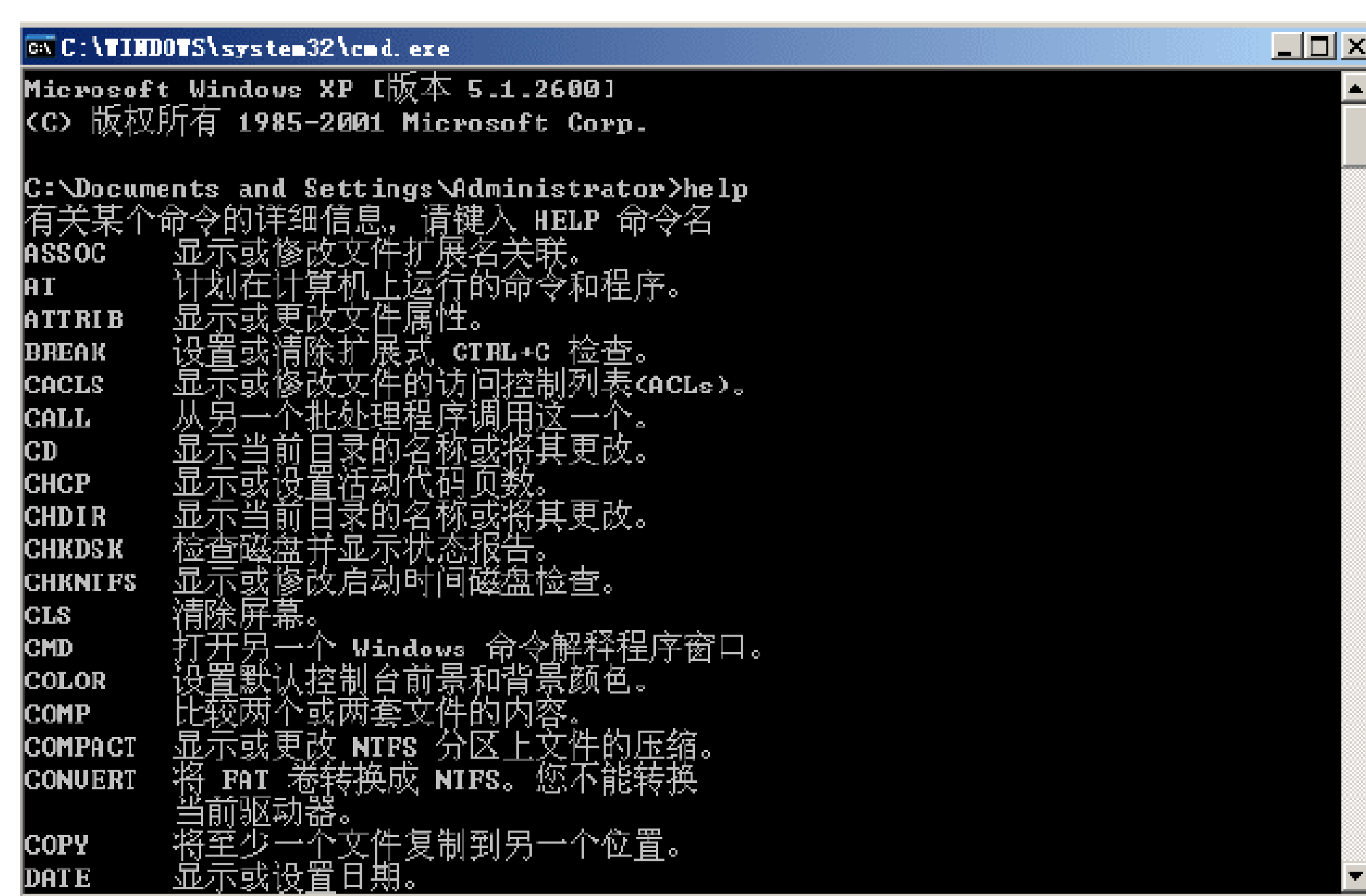


图 7.25　命令行窗口

用户通过命令行窗口输入命令即可从服务器域名地址得到服务器的 IP 地址，输入命令为“ping maile.163.com”，如图 7.26 所示。

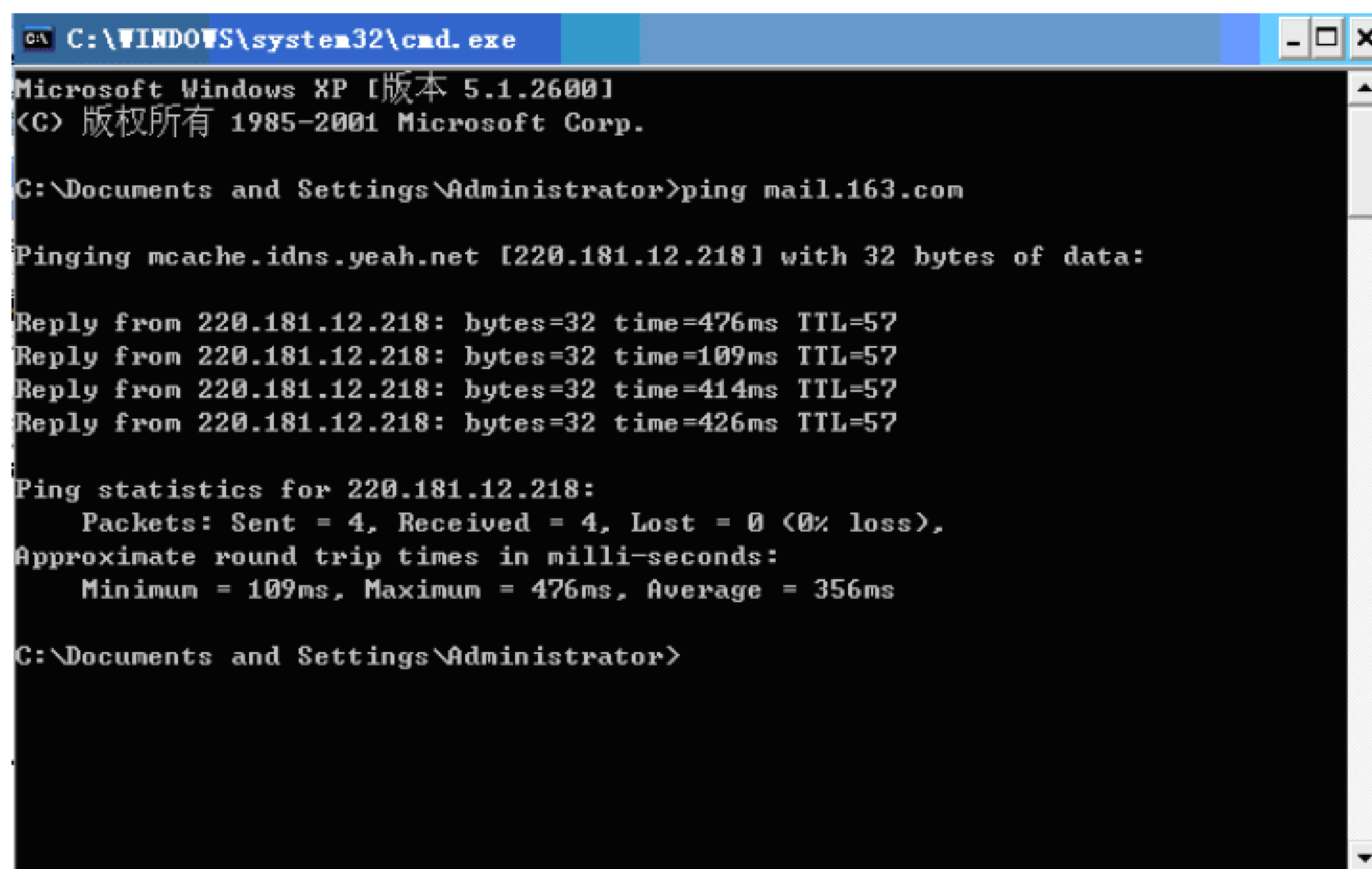


图 7.26　转换地址成功

用户将得到的转换 IP 地址输入服务器设置对话框中，便可以连接相应的服务器。当程序连接服务器成功以后，服务器会返回响应码 220。代码如下：

```
    ...                                     //省略部分代码
   char buf[4];                             //定义缓冲区
recv(s,&buf,4,0);                           //接收响应数据
if(atoi(buf)==220)                          //比较响应数据
{
  MessageBox("服务器准备就绪！");           //提示用户
}
else
  {
MessageBox("服务器启动服务失败！");
}
```

在程序中，首先使用 atoi()函数将字符类型的数字转换为整型数据，然后与服务器返回的响应码进行比较。如果相等，则表示服务器服务已经准备就绪，否则表示服务器服务启动失败。

7.3.7　构造邮件

服务器端服务成功启动以后，客户端可以将邮件发送到 SMTP 服务器，但是在邮件发送之前必须对邮件的数据进行顺序调整，以符合 SMTP 协议的规范。例如，一封正确的邮件数据格式应该如下。

```
Data:Tue,04 Feb 2009 21:18:03+0800          //邮件发送时间
From:lymlrl@163.com                         //发件人地址
To:lymlrl@126.com                           //收件人地址
Subject: This is a E-Mail                   //邮件主题
```

```
                                      //空白行
Hello lymlrl!                         //邮件内容
This is a E-mail!
```

在实例程序中，将上述邮件内容填入到响应项目中，如图 7.27 所示。用户在图 7.27 中，可以看到邮件内容与软件界面中的各个项目相对应。这里向用户讲述从邮件内容读取数据到界面各对应项目中进行显示。下面，将向用户讲述从界面中获得数据构造邮件。首先，在界面中填入相关信息，如图 7.28 所示。

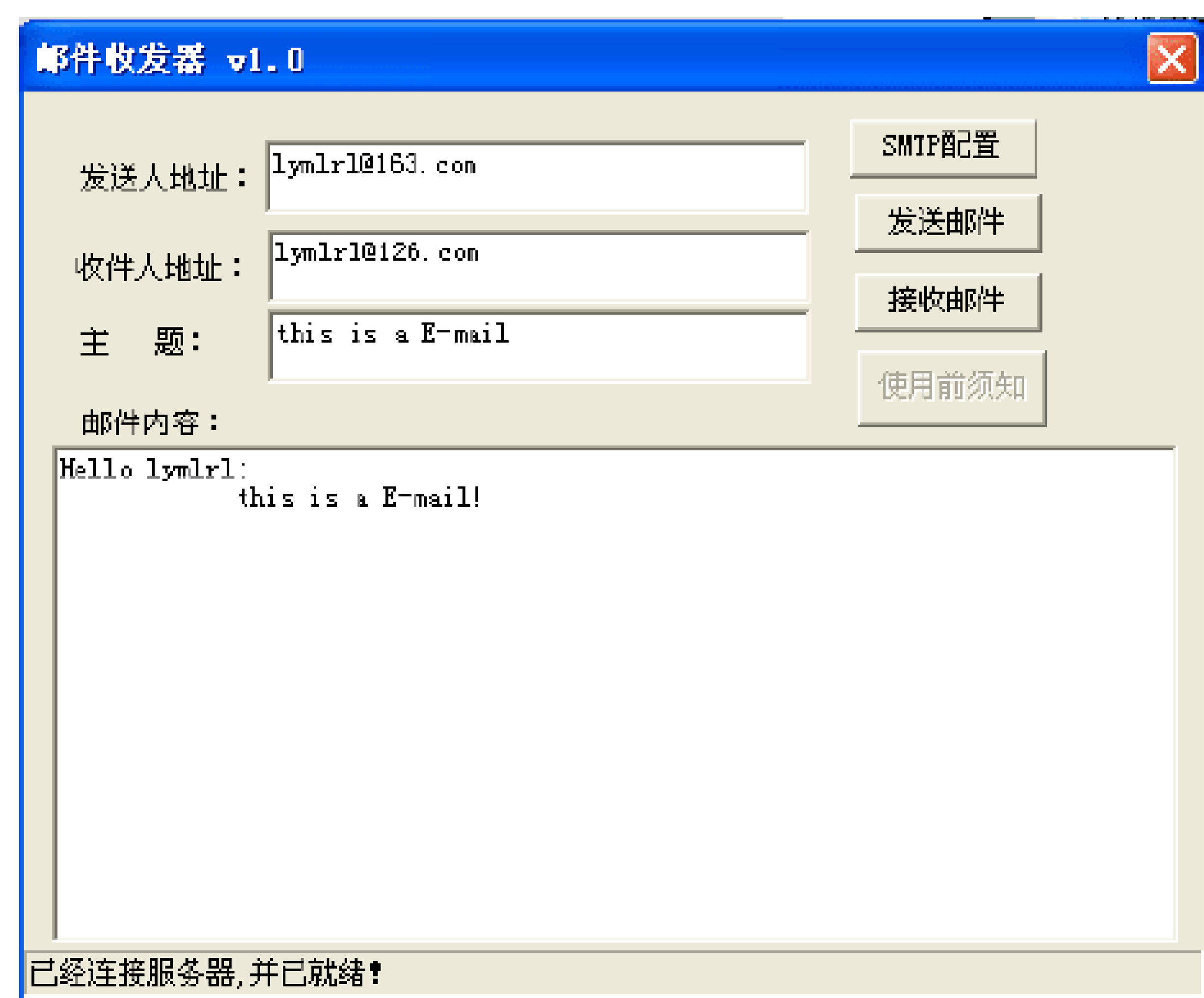


图 7.27　构建邮件

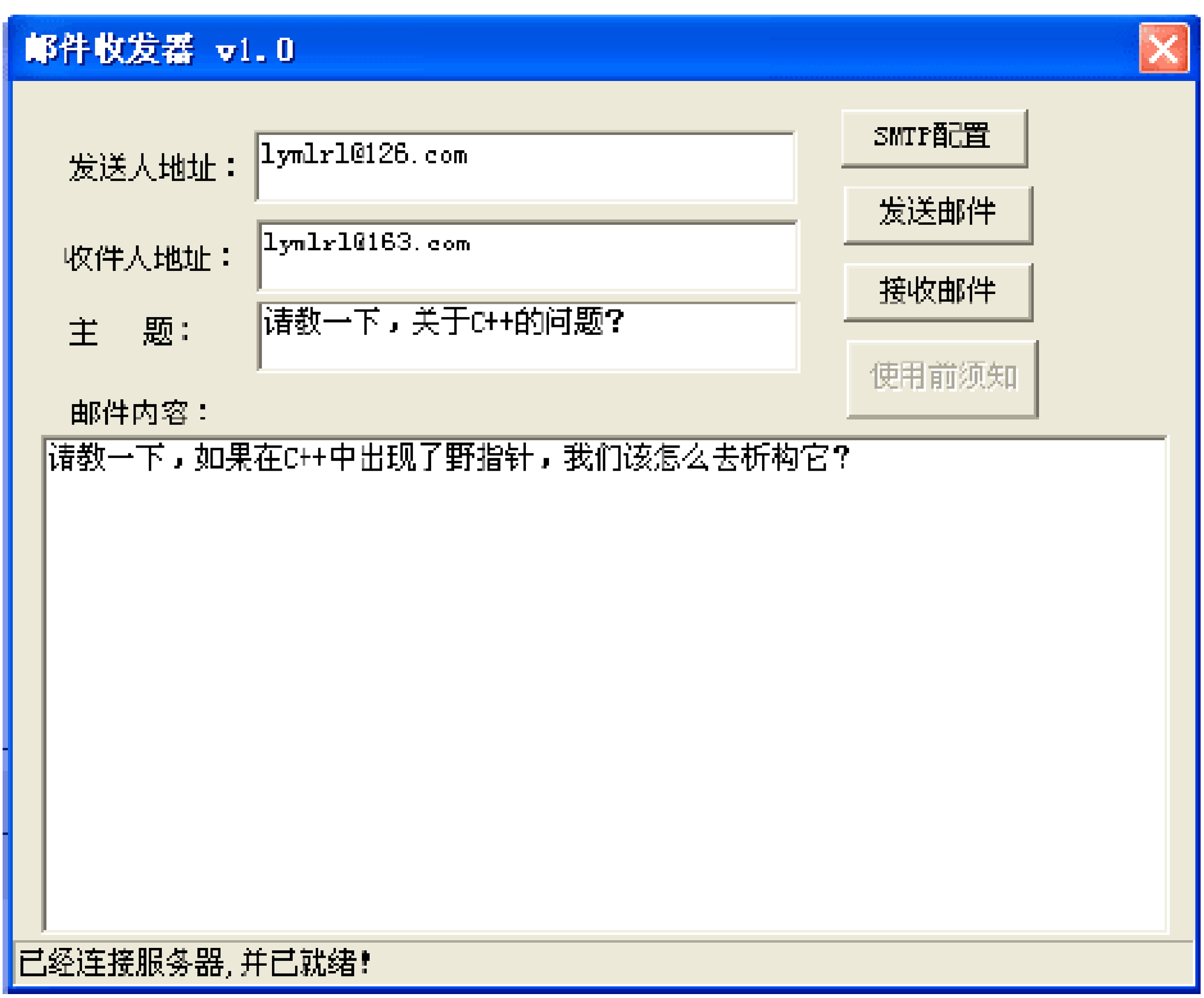


图 7.28　填入信息后的界面

然后，根据内容开始构造邮件。代码如下：

```
Data:Tue,04 Feb 2009 13:47:03+0800              //邮件发送时间
From:lymlrl@163.com                             //发件人地址
To:lymlrl@126.com                               //收件人地址
Subject: 请教一下，关于 C++的问题？             //邮件主题
                                                //空白行
                                                //邮件内容
请教一下，如果在 C++中出现了野指针，我们该怎么去析构它？
```

如果在邮件中，有多个收件人，则在邮件内容中的收件人处使用逗号将其分开即可。例如，收件人地址为“lymlrl@126.com，lymlrl@sina.com.cn，lymlrl@yahoo.com.cn”，则邮件中收件人字段应该进行修改。代码如下：

```
To:lymlrl@126.com, lymlrl@sina.com.cn,lymlrl@yahoo.com.cn //收件人地址
```

注意： 用户在构造邮件完成之后，发送到服务器之前，一定记得在邮件内容最后处加上结尾符“\0”或者是 NULL。

7.3.8　发送邮件

首先，在 VC 中为发送邮件按钮添加消息响应函数。添加方法是在双击该按钮，弹出 Add Member Function（添加函数）对话框，并显示其函数名，如图 7.29 所示。

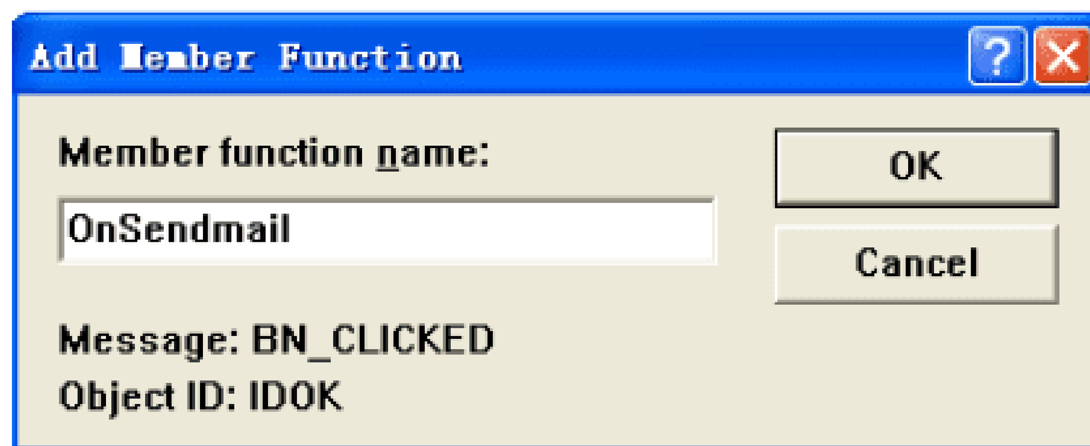


图 7.29　添加函数对话框

然后，单击 OK 按钮，VC 视图将跳转到该函数定义处。代码如下：

```
void CMyDlg::OnSendmail()
{
CString data="Data: Tue,04 Feb 2009 21:18:03+0800\r\n";//构造发送字符串
CString sender=" MAIL FROM:";
CString recver=" RCPT TO:";
CString subject=" Subject:";
CString s,r,s1;
GetDlgItem(IDC_SENDER)->GetWindowText(s);                //获取控件的内容
GetDlgItem(IDC_RECVER)-> GetWindowText(r);
GetDlgItem(IDC_SUBJECT)-> GetWindowText(s1);
GetDlgItem(IDC MAILTEXT)-> GetWindowText(mailtext);
sender+=s;                                               //添加获取内容
recver+=r;
subject+=s1;
char sengmail[]={"HELO",                                 //构造发送数组
               sender.GetBuffer(1),
               recver.GetBuffer(1),
                "DATA\r\n",                              //发送 DATA 命令
                  subject.GetBuffer(1),
                 mailtext.GetBuffer(1),
                    "QUIT\r\n",                          //退出会话
                      "\0"                               //结束符
                           }
send(s,sendmail,sizeof(sendmail),0);                     //向服务器发送邮件
                      }
```

上面的代码中，用户在构造邮件时必须在最后添加上结束符“\0”或者 NULL。DATA 命令是告知服务器准备接收客户端将要发送的邮件内容。如果邮件发送成功，服务器会返回响应码 250，所以在客户端没有接收到该响应码，则表示邮件发送未完成，继续执行发送。否则，表示发送完成。为了让用户知道当前发送的状态，在状态栏中显示该状态信息即可。代码如下：

```
{
...                                  //省略部分代码
char buf[4];                         //定义缓冲区
recv(s,buf, 4,0);                    //接收响应数据
if(buf!=NULL)                        //是否接收到数据
{
  if((atoi)buf==250)
{
::SendMessage(statu,SB_SETTEXT,0,(long)"邮件发送成功！");
}
else
{
  ::SendMessage(statu,SB_SETTEXT,0,(long)"邮件发送失败！");
```

```
}
}
else
{
    ::SendMessage(statu,SB SETTEXT,0,(long)"邮件正在发送！");
      }
}
```

以上代码是发送邮件时，用户使用相关的界面显示邮件当前的发送状态。例如，状态栏等。

7.3.9　发送邮件实例

用户将前面的程序代码完善以后，便可以实现邮件的发送功能，该功能的实现是在程序界面中发送邮件按钮的消息响应函数。代码如下：

```
void CMyDlg::OnSendmail()
{
char buf[4];                                               //定义缓冲区
CString data="Data: Tue,04 Feb 2009 21:18:03+0800\r\n";//构造发送字符串
CString sender=" MAIL FROM:";
CString recver=" RCPT TO:";
CString subject=" Subject:";
CString s,r,s1;
GetDlgItem(IDC_SENDER)->GetWindowText(s);                  //获取控件的内容

GetDlgItem(IDC RECVER)-> GetWindowText(r);
GetDlgItem(IDC SUBJECT)-> GetWindowText(s1);
GetDlgItem(IDC_MAILTEXT)-> GetWindowText(mailtext);
sender+=s;                                                 //添加获取内容
recver+=r;
subject+=s1;
char sengmail[]={"HELO",                                   //构造发送数组
               sender.GetBuffer(1),
              recver.GetBuffer(1),
                "DATA\r\n",                                //发送 DATA 命令
                  subject.GetBuffer(1),
                 mailtext.GetBuffer(1),
                    "QUIT\r\n",                            //退出会话
                      "\0"                                 //结束符
                           }
send(s,sendmail,sizeof(sendmail),0);                       //向服务器发送邮件
recv(s,buf, 4,0);                                          //接收响应数据
if(buf!=NULL)                                              //是否接收到数据
{
  if((atoi)buf==250)
{
::SendMessage(statu,SB_SETTEXT,0,(long)"邮件发送成功！");
}
else
{
  ::SendMessage(statu,SB SETTEXT,0,(long)"邮件发送失败！");
}
}
else
{
```

```
        ::SendMessage(statu,SB_SETTEXT,0,(long)"邮件正在发送！");
    }
}
```

运行上面的代码，如果邮件发送成功，则在状态栏上显示“邮件发送成功”字样，如图 7.30 所示。

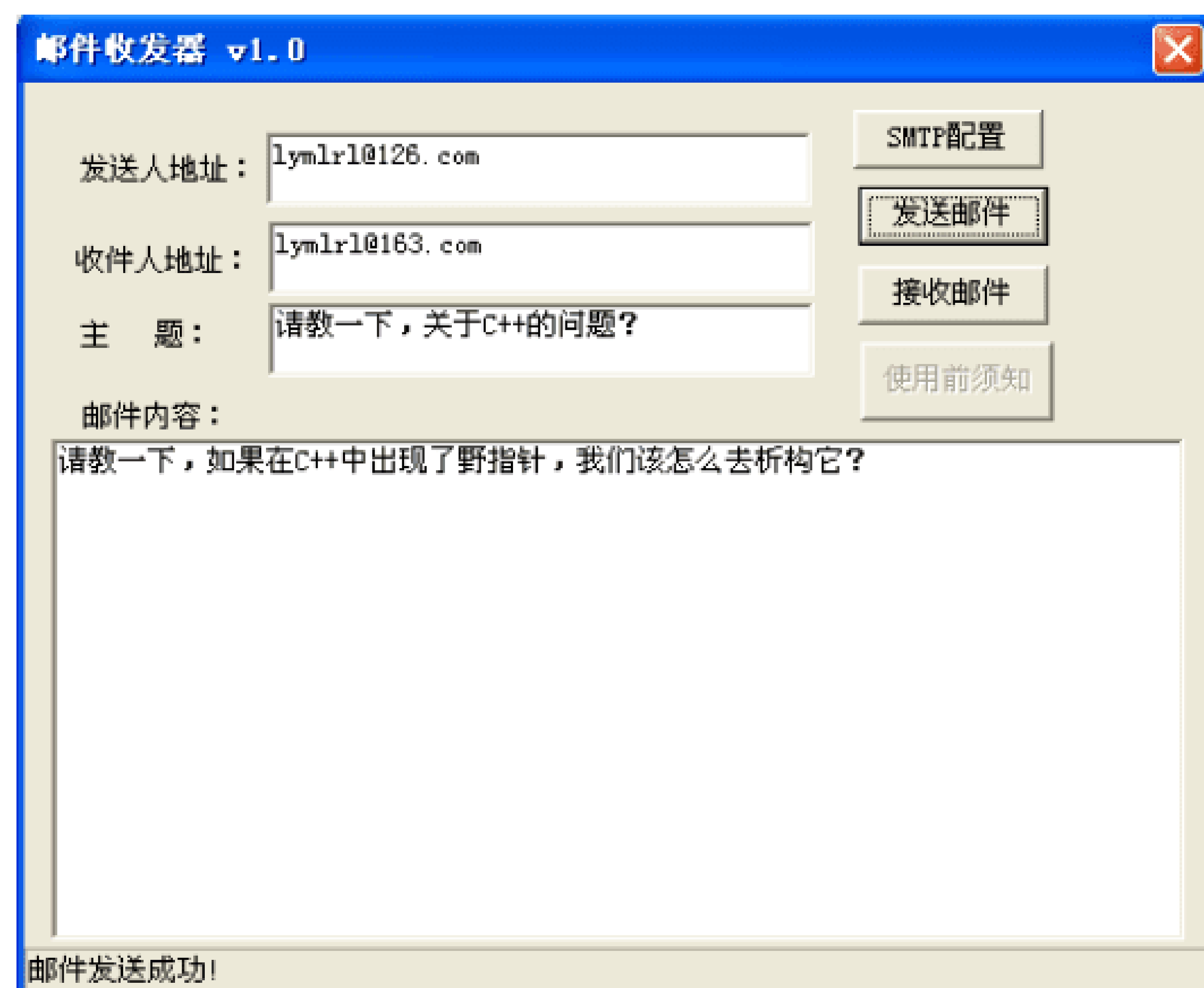


图 7.30　邮件发送成功

在本节中，向用户再次讲述了邮件的构造方法以及如何将界面控件中的内容组合成一封规范的邮件。根据 SMTP 服务器返回的响应码，判断邮件的发送状态和更新界面显示。通过本节的学习，用户能够掌握邮件发送的原理以及方法。在该实例中，其具体代码请用户参考随书光盘。

注意：读者在学习本节实例程序时，应该结合光盘中的程序代码和书中的相关说明。否则，读者学习起来将非常费力。同时，读者也可以将实例程序进行修改，满足自己的要求，使学习效果达到最佳。

7.4　接 收 邮 件

用户接收邮件是通过 POP3（接收邮件服务器）协议完成的。一般情况下，客户端通过向服务器发送相应的 POP3 命令获取邮件。服务器接收到命令以后，会将数据按照 E-Mail 的数据格式整理邮件，然后将邮件发送到客户端进行解析、显示。在本节中，将向用户讲解 POP3 命令等相关知识。

7.4.1　POP3 简介

一般，用户接收邮件是通过向 POP3（接收邮件）服务器发送命令获取的。具体发送命令的步骤与 SMTP 协议一样，所以在这里不再赘述，如有不清楚的地方请用户复习服务器前面的知识。在本节中，将向用户介绍部分 POP3 命令以及编程实现接收邮件功能。

1．POP3命令

POP3 通信方式与 SMTP 一样，使用标准命令与服务器进行数据交换。POP3 协议中还规定了标准端口为 110 号端口。POP3 标准命令如表 7.7 所示。

表 7.7　部分POP3 标准命令

命　　令	意　　义
QUIT	终止与服务器会话
STAT	提供信箱大小
LIST	获取邮件大小
USER	客户端发送账号信息到服务器验证
PASS	客户端发送密码信息到服务器验证
TOP	取出第 M 封邮件信头和邮件内容的前 N 行
DELE	删除第 N 封邮件
RSET	复位 POP3 会话
RETR	取出第 N 封邮件

在上表中列出了 POP3 的相关命令，下面将对其中的命令进行详解。

- 命令 QUIT 的作用是终止与服务器的会话连接。格式如下：

```
QUIT
```

该命令如果发送到服务器执行成功，服务器则会返回 OK，表示服务器同意客户端退出对话。

- 命令 STAT 的作用是请求服务器信箱的大小信息。
- 命令 LIST 可以获取指定邮件的大小信息。如果不带任何命令参数，则服务器会返回所有邮件的大小。格式如下：

```
LIST            //客户端发送命令 LIST
1 1024          //表示第一封邮件的大小
2 2048          //表示第二封邮件的大小
...
```

注意：格式中的序号表示邮件的序列号，紧跟后面的数字表示该邮件的大小信息。使用该命令获得的邮件列表序号是从 1 开始的。

- 命令 USER 将标识客户端发送的账号信息。格式如下：

```
USER lymlrl
```

- 命令 PASS 将标识客户端发送的密码信息。格式如下：

```
PASS lwlwlw
```

- 命令 TOP 表示将取出指定邮件的信头和其邮件内容的前 N 行。例如，用户需要取出第一封邮件的前两行内容，则发送 TOP 命令到服务器即可。代码如下：

```
CString str("TOP 1 2\r\n");                    //构造命令字符串
send(s,str.GetBuffer(1),sizeof(str),0);      //发送命令到服务器
```

- 命令 DELE 表示对邮件进行删除操作。如果该命令配合其命令参数可以删除第 N

封邮件。例如，用户将删除第 N 封邮件，格式如下：

```
DELE N
```

- 命令 RSET 的作用是对 POP3 会话过程进行复位。
- 命令 RETR 的作用是取出第 N 封邮件。例如，用户需要取出第 N 封邮件。格式如下：

```
RETR N
```

当客户端发送该命令以后，服务器会返回被请求邮件的全部内容（包括邮件头和邮件内容）。

如果服务器成功接收到 POP3 命令之后，都会返回相应的请求数据到客户端。返回的数据格式如下：

```
OK
服务器将返回相应的数据
```

2. POP3会话

POP3 会话过程与 SMTP 一样，必须首先连接服务器成功以后才能进行相关操作。下面简单介绍一下 POP3 会话的过程，代码如下：

```
连接服务器
OK
USER lymlrl                        //验证用户名
OK
PASS lwlwlw                        //验证密码
OK
retr 1                             //请求第一封邮件内容
OK 服务器将第一封邮件内容发送到客户端
Quit                               //结束会话
OK
```

上面代码中，单行为客户端操作行为，双行为服务器操作行为。该会话过程是一个交互式的问答过程。用户将上述会话过程使用编程方法来实现，代码如下：

```
...                                //省略部分代码
CString user="user lymlrl\r\n";    //构造命令信息
CString pass="pass lwlwlw\r\n";
CString retr="retr 1\r\n";
CString quit="quit\r\n";
char sendmsg[]={user.GetBuffer(1),
                pass.GetBuffer(1),
                  retr.GetBuffer(1),
                    quit.GetBuffer(1)
                                     }
send(s,sendmsg,sizeof(sendmsg),0); //发送命令数组
```

在代码中，首先定义命令字符串，然后再将这些命令字符串组合到一个字符数组中，并发送到服务器执行。

注意： 因为 POP3 的工作方式与 SMTP 相似，所以在本章中不再向读者继续讲解关于 POP3 的其他知识。如果用户需要具体了解，请参考其他书籍。

7.4.2　接收邮件实例界面

在本章实例程序中，需要为工程添加接收邮件所使用的对话框，并且在 VC 编译环境中对该对话框界面进行设置。本节主要讲述界面的设计以及代码实现方法。

1. 添加接收邮件界面

接收邮件与发送邮件一样，通过向服务器发送相关 POP3 命令以获取相应的邮件。本章中，用户需要为工程“邮件收发器”添加相应的界面以便实现接收邮件功能。添加邮件接收界面的步骤如下：

（1）打开 VC 资源管理器，右击 Dialog，弹出快捷菜单，如图 7.31 所示。

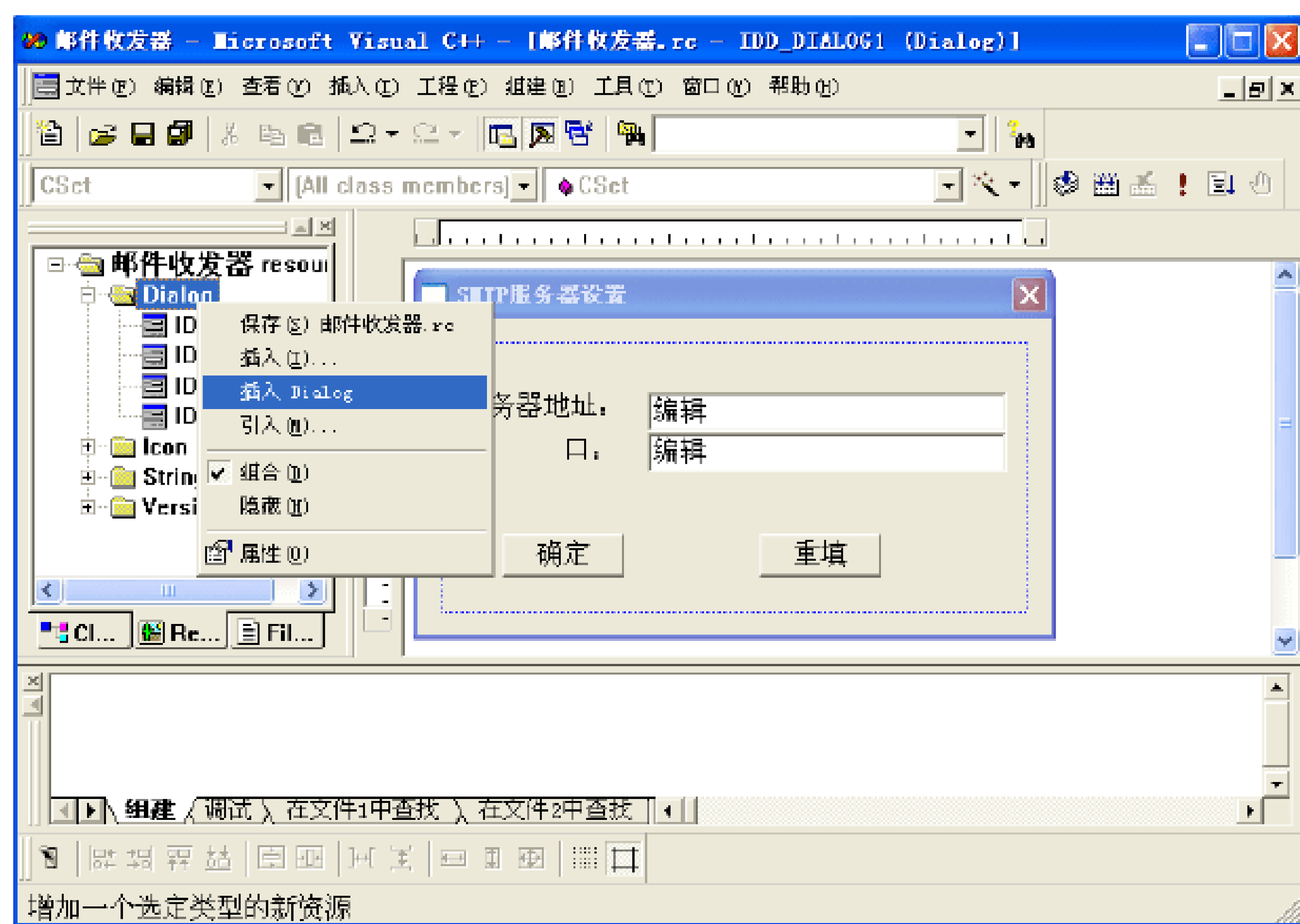


图 7.31　添加对话框资源

当然，用户除了使用上述添加对话框的方法之外，还可以使用窗口中的“插入”菜单。

（2）选择“插入 Dialog”命令，在 VC 工程中添加一个新建对话框。界面如图 7.32 所示。

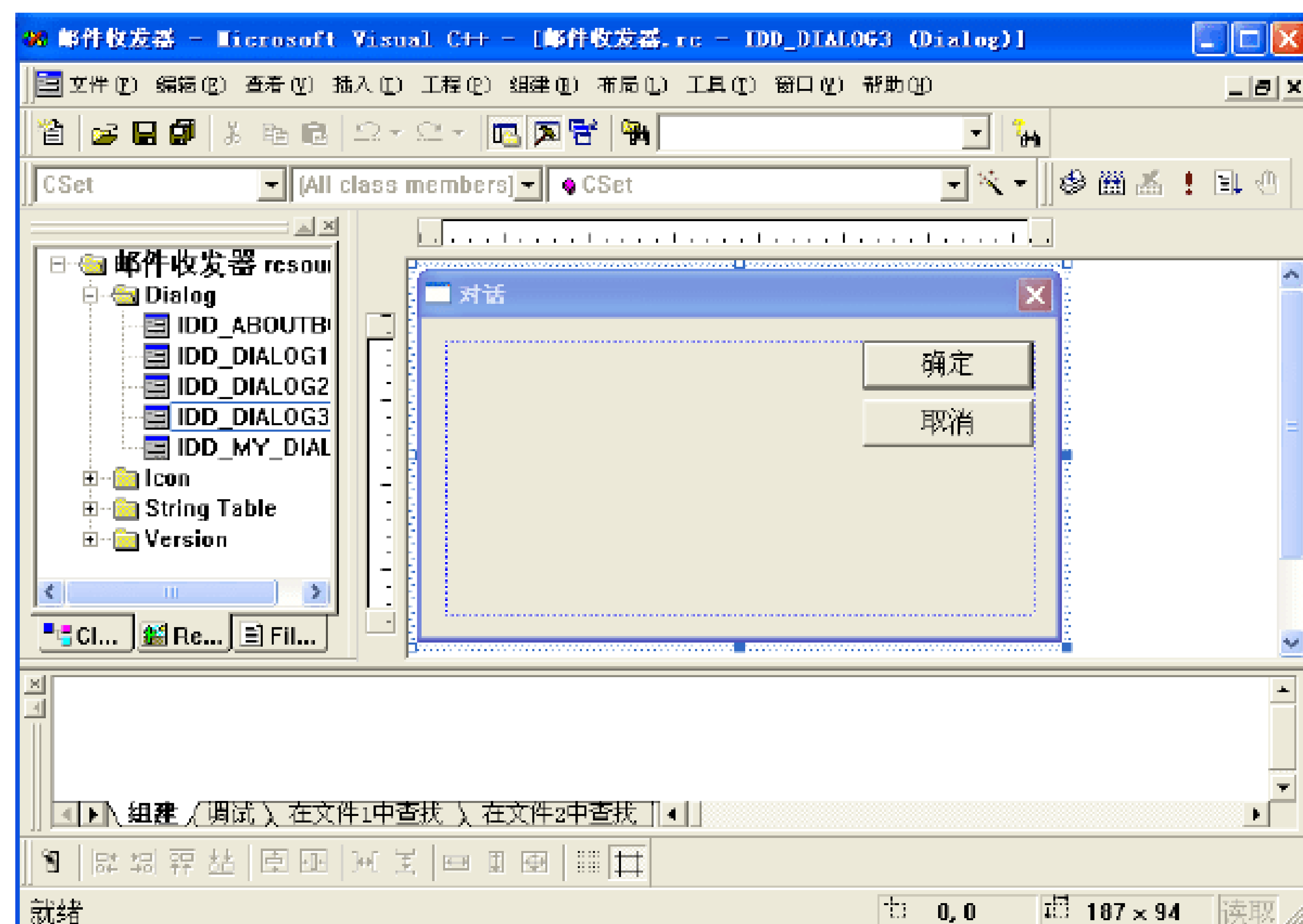


图 7.32　新建对话框界面

（3）用户可以向新建对话框界面中添加控件，以满足接收邮件功能的需要。本实例中，接收邮件的界面效果如图 7.33 所示。

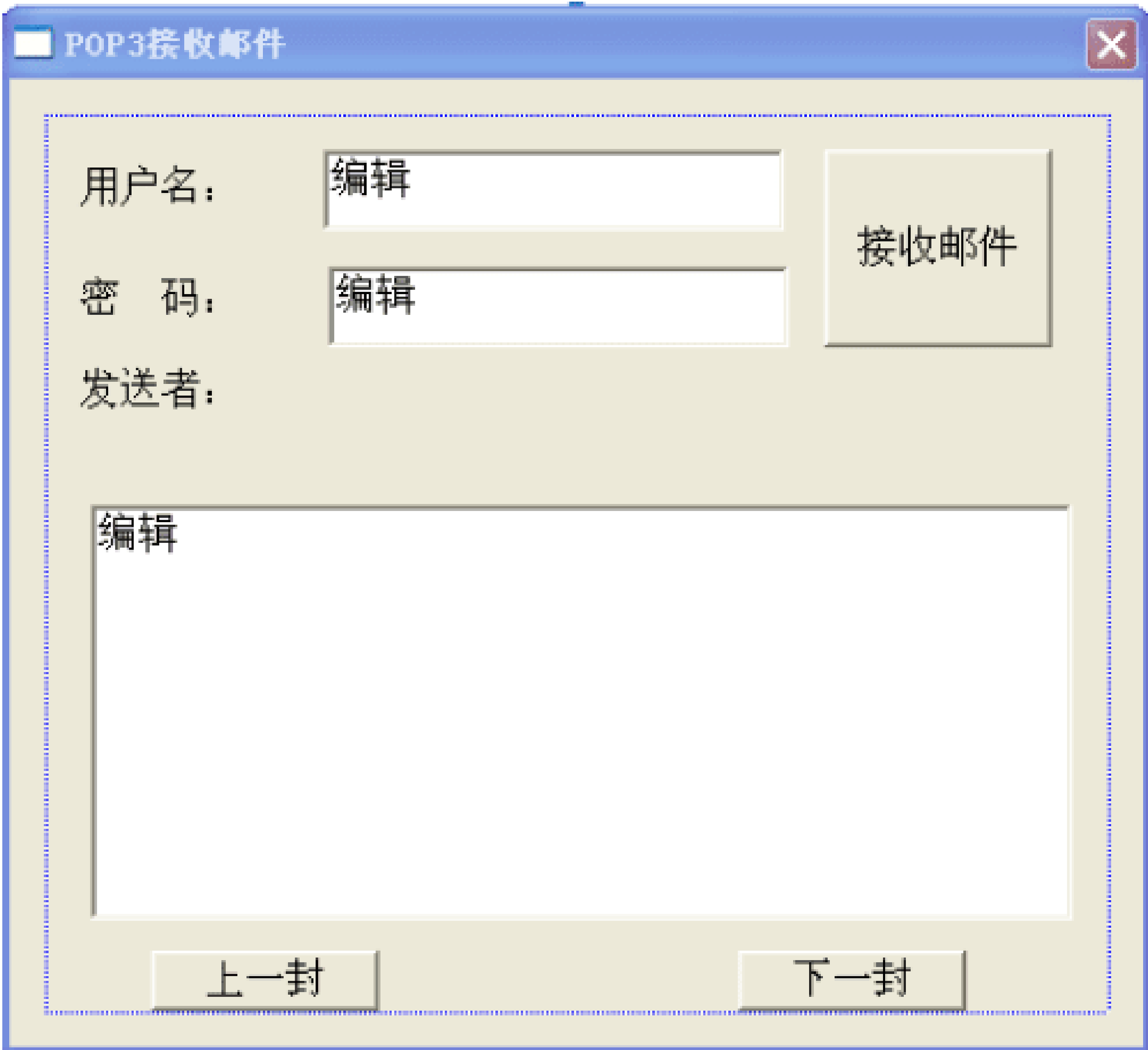


图 7.33　接收邮件界面

在图 7.33 所示界面中，所添加的控件列表如表 7.8 所示。

表 7.8　添加控件列表

控　件　ID	控 件 类 型	控 件 作 用
IDC_STATIC	静态文本控件	标识“发送者”
IDC_NAEM	静态文本控件	显示发送者邮件地址
IDC_EDIT1	编辑框控件	显示获取到的邮件内容
IDC_SHANG	按钮控件	显示上一封邮件
IDC_NEXT	按钮控件	显示下一封邮件
IDC_RECV	按钮控件	接收邮件操作
IDC_ZHANGHU	编辑框控件	输入用户名
IDC_PASS	编辑框控件	输入密码

通过以上 3 个步骤，在工程项目中已经成功添加了接收邮件的对话框。现在，用户可以为刚添加的对话框指定一个新类，按下键盘上的 Ctrl+W 组合键，弹出 New Class（添加新类）对话框，如图 7.34 所示。

用户在该对话框中，为添加的接收邮件对话框指定类名为 CRecv，其他选项设为默认。单击 OK 按钮返回主界面，用户可以在 VC 资源管理器的类视图列表中看到该类的相关文件。打开该类头文件定义变量。代码如下：

```
class CRecv : public CDialog
{
public:
CString mailadd;                //定义发送者邮件地址
CString mailtext;               //定义邮件内容
CString name;                   //定义用户名
CString pass;                   //定义密码
```

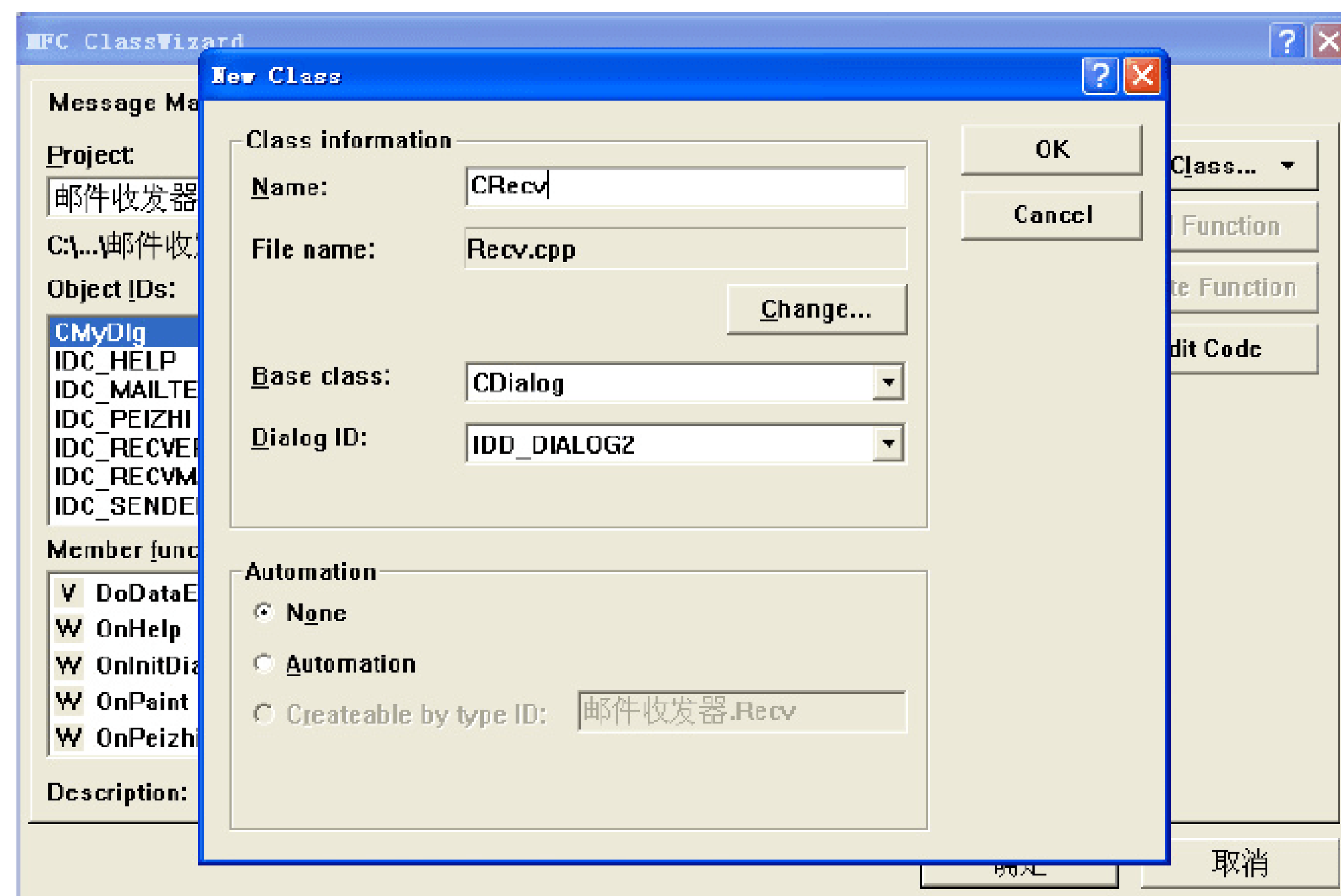


图 7.34　添加新类对话框

```
CString netadd;                     //定义服务器地址
CString port;                       //定义服务器端口
...                                 //省略部分代码
}
```

在 CRecv 类中，定义用户名、密码、邮件内容等字符串变量，并且将保护属性设置为公有属性。

7.4.3　使用接收邮件对话框

如果用户需要在工程主对话框类 CMyDlg 中使用 CRecv 类，还必须在头文件“邮件收发器 Dlg.h”中包含该类头文件 Recv1.h。代码如下：

```
...                                 //省略部分代码
#include "Set.h"                    //包含 CSet 类头文件
#include "Recv1.h"                  //包含 CRecv 类头文件
...                                 //省略部分代码
```

然后，在 CMyDlg 类中定义 CRecv 类对象。代码如下：

```
class CMyDlg : public CDialog
{
...                                 //省略部分代码
protected:
    CSet set;                       //SMTP 设置对话框类
    CRecv recvdlg;                  /接收邮件对话框类
     }
```

在接收邮件按钮的响应函数 OnRecvmai()中，使用 CRecv 类对象调用接收邮件对话框。代码如下：

```
void CMyDlg::OnRecvmail()
{
   ...                              //省略部分代码
```

```
    recvdlg.DoModal();            //使用接收邮件对话框
}
```

用户在编译器中，编译运行以上代码，然后单击“接收邮件”按钮，将弹出接收邮件模式对话框，如图 7.35 所示。

在程序中为了防止用户使用不当，使其发生错误，所以显示该对话框时，应该使“接收邮件”按钮、显示邮件内容的编辑框等主要控件被禁用。例如，当用户输入用户名及密码以后，使“接收邮件”按钮可用。当用户浏览第二封邮件之后，使“上一封”按钮处于可用状态。初始化界面代码如下：

```
BOOL CRecv::OnInitDialog()
{
    CDialog::OnInitDialog();
    GetDlgItem(IDC_RECV)->EnableWindow(false);     //禁用各个按钮
    GetDlgItem(IDC_EDIT1)->EnableWindow(false);
    GetDlgItem(IDC_SHANG)->EnableWindow(false);
    GetDlgItem(IDC_NEXT)->EnableWindow(false);
    return TRUE;
}
```

运行以上代码，对话框初始化界面如图 7.36 所示。



图 7.35　模式对话框



图 7.36　界面初始化

注意：在工程中实现邮件接收功能均在“POP3 接收邮件”对话框中进行实现。

7.4.4　接收邮件

在实例程序中，用户使用接收邮件对话框类 CRecv 对邮件进行接收。但是，在该对话框中还需要进行一些功能上的实现。例如，为界面中的按钮添加消息响应函数等操作。

1．添加消息响应函数

首先，为接收邮件按钮添加消息响应函数，添加方法如图 7.37 所示。该函数的作用是

根据用户输入的用户名以及密码获取 POP3 服务器中对应的邮件。

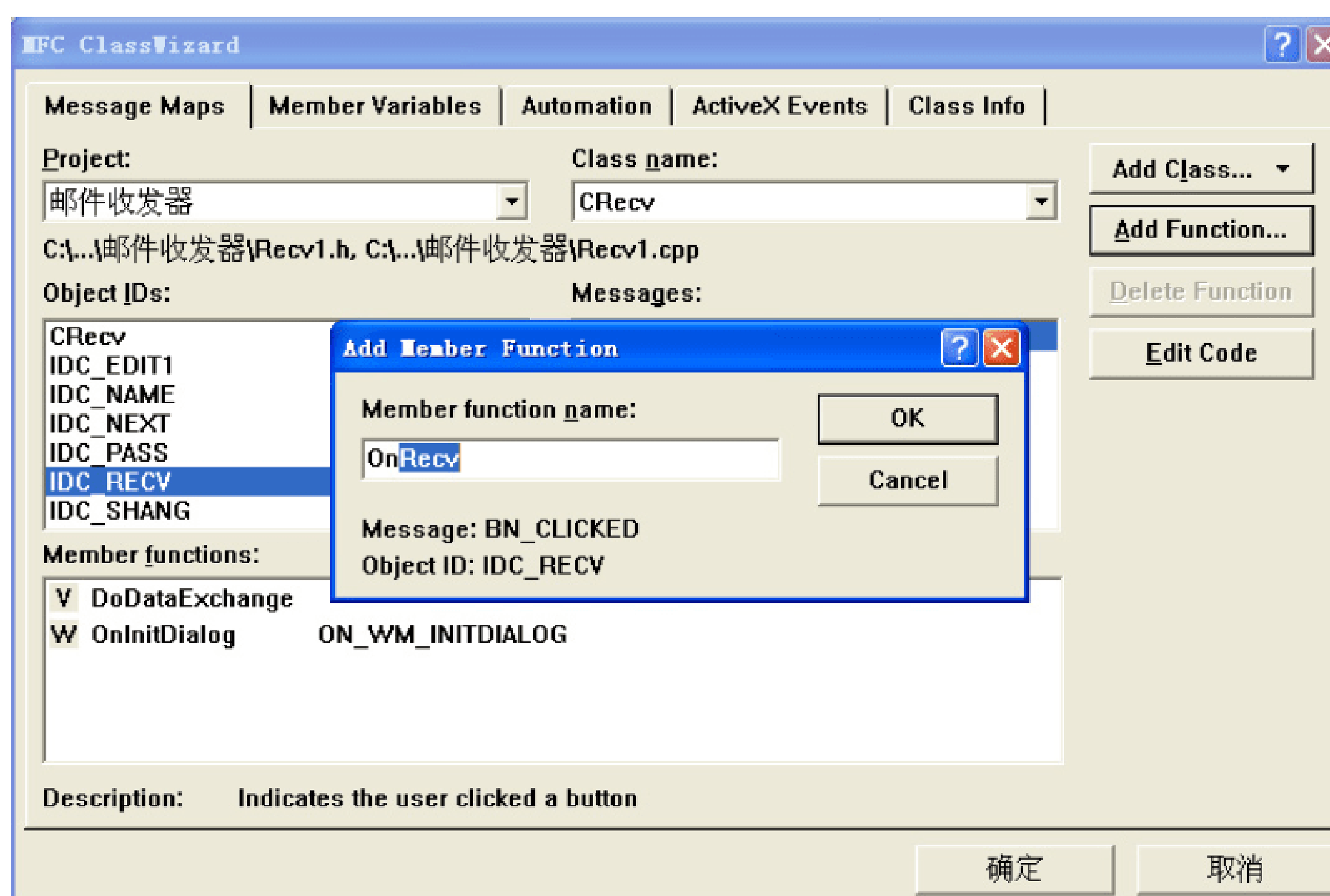


图 7.37　添加“接收邮件”按钮消息响应函数

注意：用户可以在 Member function name 编辑框中修改响应函数名，在本实例中，将该函数名设置为 OnRecv。然后单击 OK 按钮，返回 VC 主界面。

然后，分别为“上一封”按钮和“下一封”按钮添加消息响应函数，如图 7.38 和图 7.39 所示。这两个函数的作用分别是方便用户阅读上一封邮件和阅读下一封邮件。

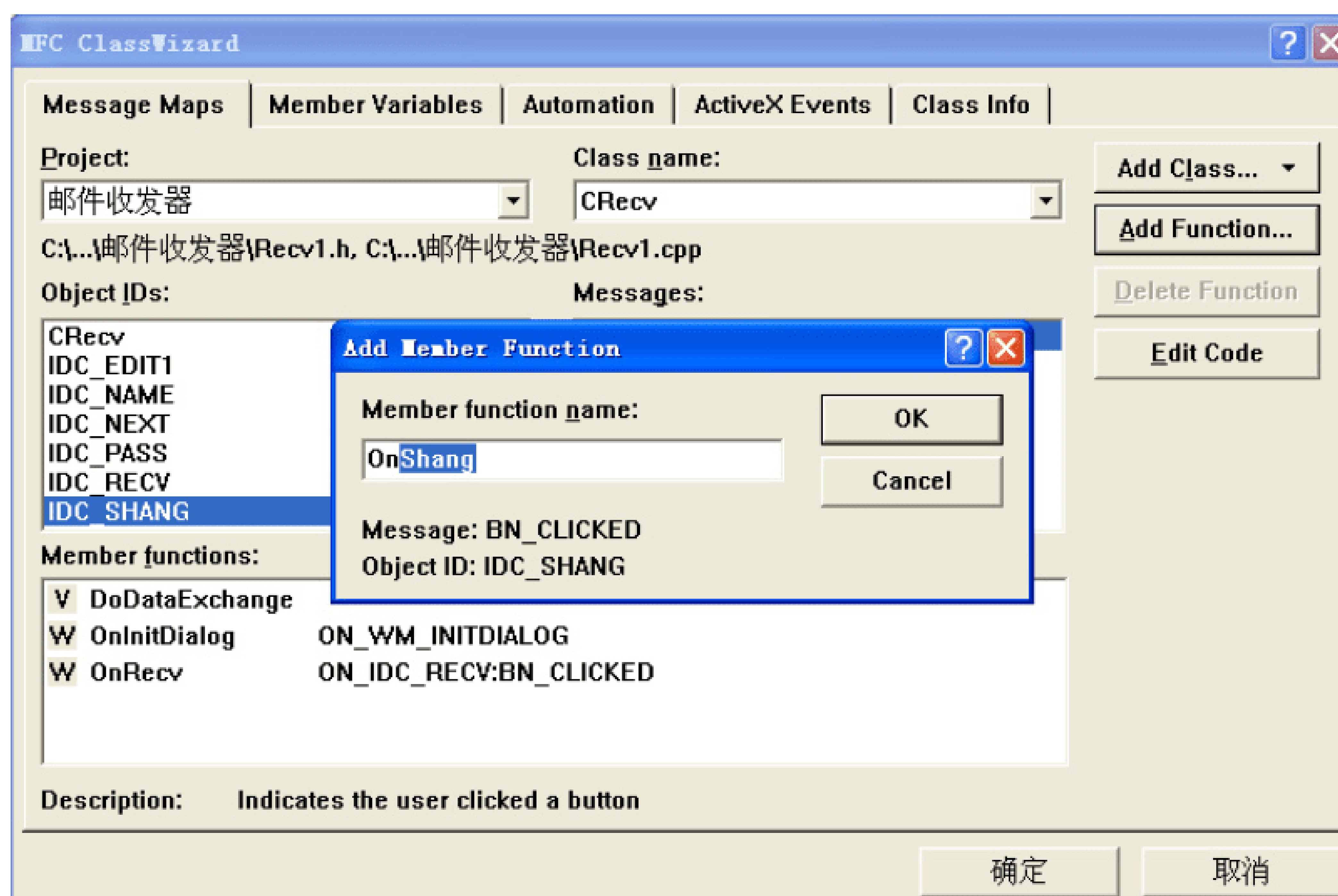


图 7.38　添加“上一封”按钮消息响应函数

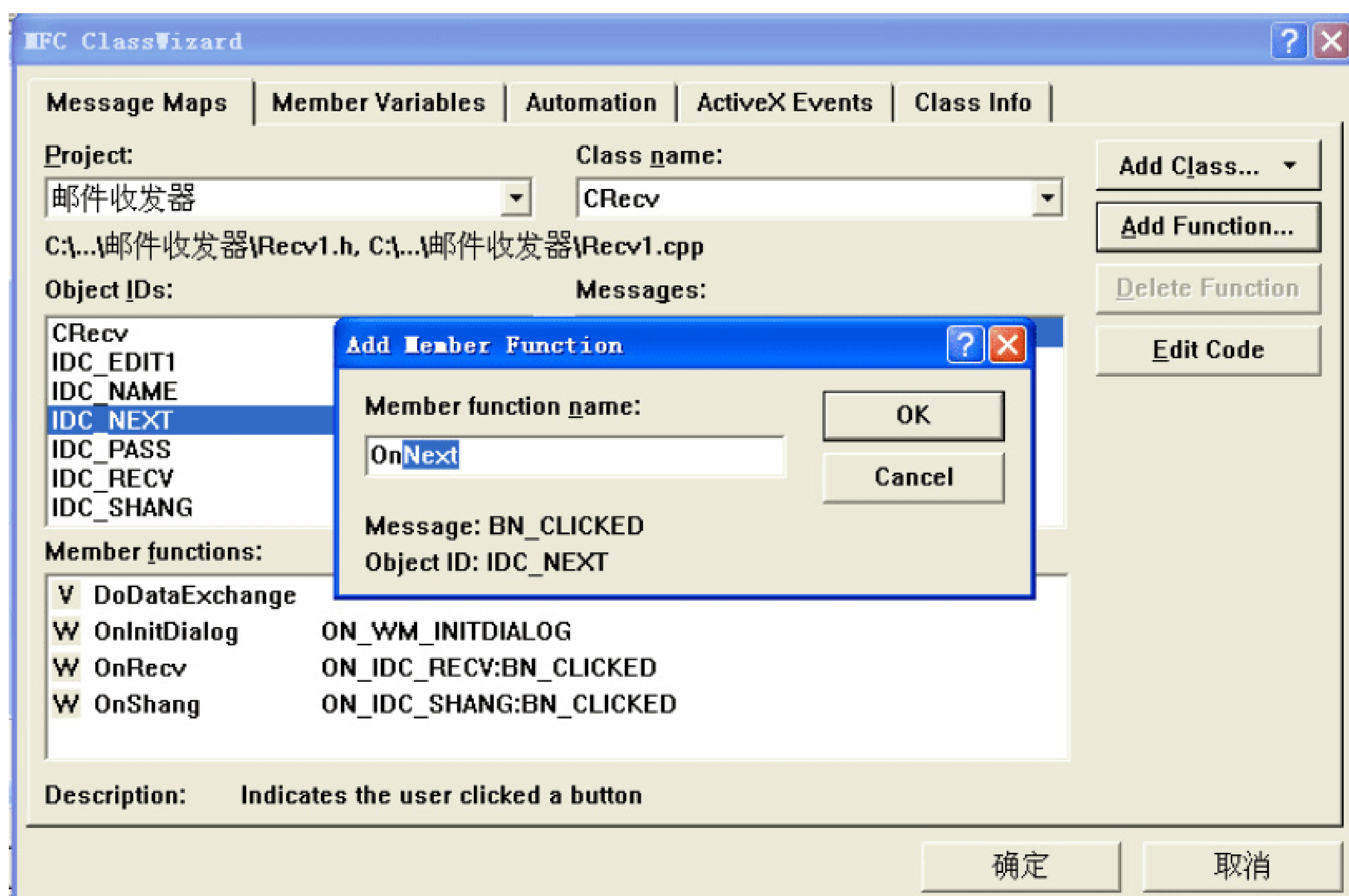


图 7.39　添加“下一封”按钮消息响应函数

注意：在本实例中，将“上一封”按钮和“下一封”按钮的消息响应函数分别命名为 OnShang()和 OnNext()。

最后，程序等待用户输入用户名以及密码结束后，应该使按钮“接收邮件”处于可用状态。所以，用户应该为密码编辑框 IDC_PASS 响应消息 EN_CHANGE，函数名为 OnChangePass()，如图 7.40 所示。

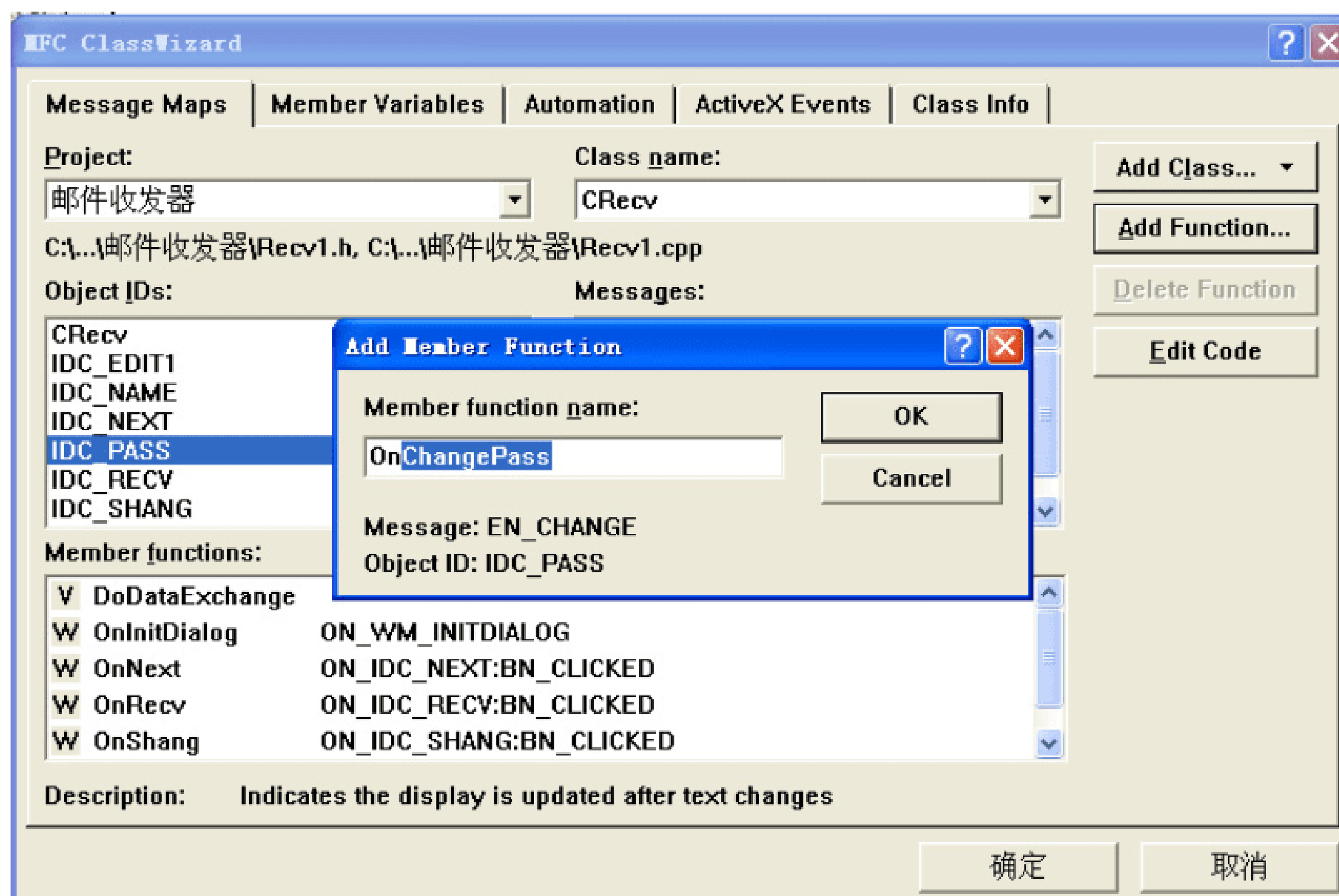


图 7.40　为 IDC_PASS 添加消息响应函数

用户在工程中添加响应函数成功以后，各个按钮的响应函数分别是 OnRecv()、

OnShang()、OnNext()和 OnChangePass()。

2．更新界面状态

当用户在密码编辑框中输入用户密码以后，“接收邮件”和“下一步”按钮应该处于可用状态。代码如下：

```
void CRecv::OnChangePass()
{
    if(i>3)                                         //表示用户至少输入 3 个字符
    {
        CString str;                                //定义字符串

        this->GetWindowText(str);                   //获取编辑框中的内容

        if(str.Find("\n"))                          //以回车键结束输入

            GetDlgItem(IDC_RECV)->EnableWindow(true);  //设置按钮可用
           GetDlgItem(IDC NEXT)->EnableWindow(true);
    }
    else
    {
        i+=1;
    }
}
```

在代码中，变量 i 的作用是记录用户输入字符的个数，该变量是在 CRecv 类中定义并在初始化函数 OnInitDialog()初始化。代码如下：

```
class CRecv : public CDialog
{
public:
    CRecv(CWnd* pParent = NULL);
int i;                                  //定义变量 i
int n;                                  //表示邮件序列号
HWND stat;                              //定义状态栏句柄
SOCKET s;                               //定义套接字句柄
...                                     //省略部分代码
}
BOOL CRecv::OnInitDialog()              //初始化函数
{
    CDialog::OnInitDialog();
    ...                                 //省略部分代码
i=0;                                    //初始化变量 i
n=0;                                    //初始化邮件序列号为 0
statu=::CreateStatusWindow(WS_CHILD|WS_VISIBLE,"接收邮件)",this->m_hWnd,
IDC_123);                               //创建状态栏
    return TRUE;
}
```

运行以上代码，用户在密码编辑框中输入密码以后，程序开始更新界面，如图 7.41 所示。



图 7.41　更新后的界面

7.4.5　实现接收邮件功能

在程序中，接收邮件功能是在接收邮件按钮的消息响应函数中实现的。该函数名为 OnRecv()，代码如下：

```
void CRecv::OnRecv()
{
   addr.sin_family=AF_INET;                          //为地址结构中的成员赋值
addr.sin_port=htons(set.m_port);
//host=::gethostbyname(set.m_severadd.GetBuffer(1));   //获取主机地址
addr.sin_addr.S_un.S_addr=inet_addr(set.m_severadd.GetBuffer(1));
                                                     //转换 IP 地址
s=::socket(AF_INET,SOCK_STREAM,IPPROTO_TCP);         //创建套接字
if(connect(s, (sockaddr*) &addr,sizeof(addr)))       //连接服务器
{
::SendMessage(statu,SB_SETTEXT,0,(long)"正在构造请求命令！");
CString str,str1;                                    //定义字符串
GetDlgItem(IDC_ZHANGHU)->GetWindowText(name);        //获取用户名
    GetDlgItem(IDC_PASS)->GetWindowText(pass);       //获取用户密码
    str.Format("USER %s",name);                      //格式化用户名命令字符串
    str+="\r\n";                                     //添加回车换行符
    str1.Format("PASS %s",pass);                     //格式化密码命令字符串
    str1+="\r\n";                                    //添加回车换行符
    str+=str1;                                       //连接两个字符串
    ::SendMessage(statu,SB_SETTEXT,0,(long)"正在发送请求命令！");
                                                     //提示用户正在发送命令
    send(s,str.GetBuffer(1),sizeof(str),0);          //发送命令字符串
    char recv[100]={0};                             //定义字符数组用于接收数据
            if(recv(s,recv,100,0))                   //接收数据
   {
     if(recv[]=='O'&&recv[1]=='K')                   //服务器应答成功
    {
    ::SendMessage(statu,SB_SETTEXT,0,(long)"服务器应答成功！");
```

```
        SendCmdAndRecv(0)                           //调用自定义函数进行命令发送
}}
  else                                              //接收失败
   {
     ::SendMessage(statu,SB_SETTEXT,0,(long)"接收失败！");
}}
  else                                              //连接失败
     {
    ::SendMessage(statu,SB SETTEXT,0,(long)"连接失败！");
          }
}
```

在上面的代码中，用户首先填充网络地址结构对象 addr，然后创建套接字对象 s，并且使用该套接字句柄进行连接服务器。如果服务器连接成功，则将构造成功的命令字符串发送到服务器执行。服务器执行成功则会返回字符串 OK，接下来程序调用自定义函数 SendCmdAndRecv()进行其他命令的发送。

7.4.6　封装客户端发送与接收功能

在本实例中，由于客户端需要发送 SMTP 命令以及接收服务器响应码和邮件数据。所以，为了方便用户在客户端编程时调用这些功能，需要自定义一个函数实现客户端发送命令与接收服务器返回的数据。本节中，将该自定义函数命名为 SendCmdAndRecv()，下面将具体介绍在实例中如何声明和实现该函数。

首先，在 CRecv 类中对自定义函数 SendCmdAndRecv()进行声明。该函数声明如下：

```
class CRecv : public Cdialog                        //CRecv 类声明
{
public:
   ...                                              //省略部分代码
     void SendCmdAndRecv (int x);                   //自定义函数
   ...                                              //省略部分代码
          }
```

上面代码的作用是在 CRecv 类中，手动添加自定义函数 SendCmdAndRecv()的方法。由于该函数方法在前面的知识中以及进行了详细的讲解，所以，在这里不再对此知识点进行讲述。自定义函数 SendCmdAndRecv()的作用是在客户端发送用户信息到服务器，并且待服务器验证成功后，客户端用于发送获取邮件的具体命令，参数 x 表示获取邮件的序列号码。

然后，在 CRecv 类中编写代码实现自定义函数的功能。代码如下：

```
void CRecv::SendCmdAndRecv(int x)
{
n=x;                                                //将参数值赋予该类中变量 n
CString str;                                        //定义字符串用于构造命令
  char recvdata[1024]={0},ch[1024],ch2[1024];       //接收数据数组
  str.Format("retr d%",n);                          //格式化字符串
  str+="\r\n";                                      //添加回车换行符
   send(s,str.GetBuffer(1),sizeof(str),0);          //发送获取邮件命令
   if(recv(s,recvdata,1024,0))                      //接收数据
    {
```

```
    if(recvdata[]=='O' && recvdata[1]=='K')          //接收成功
      {
          recvdata[sizeof(recvdata)+1]="\0";         //将字符数组转换为字符串
          maliadd=recvdata[];
 while(i=mailadd.Find("from:")!=-1 || i< mailadd.Find("\r\n"))
                                                     //循环查找
{
  ch[i++]=recvdata[i++];                             //拷贝字符
   }
GetWindowText(IDC_NAME)->SetWindowText(&ch);         //设置发送者邮件地址
if(i=mailadd.Find("\n")!=-1)                         //查找空行
{
ch2[i++]=recvdata[i++];                              //拷贝字符
GetWindowText(IDC_EDIT1)->SetWindowText(&ch);        //显示邮件内容
        }
        }
else
  {
     MessageBox("接收失败！");
            }
        }
}
```

该自定义函数实现了发送 RETR 命令，同时接收服务器返回的请求邮件并且将邮件内容显示在界面中。在程序中，发送相关命令到服务器，如果服务器执行命令成功，则返回相应的邮件，客户端接收到该邮件数据后，在数据中查找发件人的邮件地址以及邮件内容等进行显示。该函数对于本章实例中接收邮件功能的实现非常重要，请用户仔细查看代码并学习。运行以上程序后，用户接收邮件，如图 7.42 所示。



图 7.42　接收邮件

用户通过代码运行后的效果，可以知道自定义函数 SendCmdAndRecv()不但可以发送命令，还可以接收邮件内容并显示。当用户查看当前邮件以前或以后的邮件时，同样需要使用到该函数。

7.4.7　显示邮件数据

当用户浏览完当前邮件以后，如果想继续浏览下一封邮件则单击“下一封”按钮即可。但是当用户浏览的当前邮件已经是邮箱中最后一封邮件时，程序将返回一个错误信息给用户。例如，显示“指定邮箱中已经没有可供显示的邮件了”等信息。“下一封”按钮的消息响应函数名称为 CRecv::OnNext()，在该函数编写程序实现用户查看下一封邮件功能。代码如下：

```
void CRecv::OnNext()
{
    n+=1;                                //使当前邮件序列号自动加 1，指向下一封
    this->SendCmdAndRecv(n);             //调用自定义函数发送相关命令并且显示邮件
    if(!GetDlgItem(IDC_SHANG)->IsVisible())     //获得上一封按钮的当前状态
{
GetDlgItem(IDC_SHANG)->EnableWindow(true);      //显示该按钮
}
}
```

该按钮的响应函数主要是实现用户从 POP3 服务器上获取当前邮件的下一封邮件内容，并将其显示到程序界面上。函数 IsVisible()的作用是查看对象当前状态是否可用，在本程序中使用该函数是为了获得“上一封”按钮的显示状态以便确定按钮可用或禁用。

在 VC 主界面中，保存该响应函数代码并且运行，用户单击“下一封”按钮以后，程序调用自定义函数 SendCmdAndRecv()发送客户端请求并且接收显示相应邮件内容。否则，提示用户发生错误，如图 7.43 所示。



图 7.43　“下一封”按钮响应

如果用户需要浏览上一封邮件，则单击“上一封”按钮进行浏览即可。该按钮的实现原理与前面所讲的下一封按钮一样。上一封按钮的消息响应函数名称为 CRecv::OnShang()，作用是显示当前邮件的前面一封邮件内容。代码如下：

```
void CRecv::OnShang()
```

```
{
    n=n-1;                                      //当前邮件序列号减 1
if(n==0)                                        //如果当前邮件已经是第一封邮件
{
  MessageBox("当前邮件已经是第一封邮件");
}
else
   {
     this->SendCmdAndRecv(n);                   //调用自定义函数发送相关命令并且显示该邮件
}
}
```

在代码中，使用当前邮件的序列号减去 1 获得前一封邮件的序列号。如果该序列号为 0，则提示用户当前邮件是第一封邮件，否则调用自定义函数 SendCmdAndRecv()执行按钮功能。如果当前邮件之前已经没有任何邮件，那么程序将会弹出消息框提示用户发生错误，如图 7.44 所示。



图 7.44　浏览上一封邮件发生错误

如果用户浏览的当前邮件之前还有其他邮件，则程序将会显示邮件的内容以及邮件发送者的邮件地址。

到这里，用户已经编写完本实例中重要代码，实现了程序的基本功能。如果用户需要进一步学习邮件收发的相关知识或者实现更为接近实用的邮件收发程序，则可以对本实例中的界面或者代码进行修改完善，例如，添加发送或者接收带附件功能的程序代码。这样对用户学习邮件收发原理会起到推波助澜的作用。

7.4.8　代码分析

在这里，本节将对自定义函数 SendCmdAndRecv()的功能代码进行分析。该自定义函数的原型如下：

```
void SendCmdAndRecv(int x)
```

该函数的作用在 7.4.2 节中已经向用户讲解。如果函数调用成功，则将在程序窗口相

应的控件中进行显示。该函数含有一个参数 x，表示要获取的邮件序列号。用户调用函数时需要指定该参数，一般情况下，当程序启动时，参数 x 均从 1 开始计数。函数将“RETR”命令与参数 x 格式化后再发送到服务器。代码如下：

```
...                                                      //省略部分代码
n=x;                                                     //将参数值赋予该类中变量 n
CString str;                                             //定义字符串用于构造命令
  char recvdata[1024]={0},ch[1024],ch2[1024];            //接收数据数组
  str.Format("retr d%",n);                               //格式化字符串
  str+="\r\n";                                           //添加回车换行符
   send(s,str.GetBuffer(1),sizeof(str),0);               //发送获取邮件命令
...                                                      //省略部分代码
```

在代码中，首先定义字符串对象，然后根据参数将字符串格式化以后，通过函数 send()发送到服务器。在格式化命令字符串时，一定需要记住在命令最后必须加上符号“\r\n”，因为这样才能使服务器知道客户端发送的命令结束。

如果服务器接收并执行命令以后，将返回客户端所请求的相应邮件内容。这时，客户端程序调用函数 recv()进行接收并且在代码中实现数据分析。代码如下：

```
...                                                      //省略部分代码
if(recv(s,recvdata,1024,0))                              //接收数据
    {
    if(recvdata[]=='O' && recvdata[1]=='K')              //执行成功
       {
          recvdata[sizeof(recvdata)+1]="\0";             //将字符数组转换为字符串
          maliadd=recvdata[];
 while(i=mailadd.Find("from:")!=-1 || i< mailadd.Find("\r\n"))//循环查找
{
  ch[i++]=recvdata[i++];                                              //拷贝字符
   }
GetWindowText(IDC_NAME)->SetWindowText(&ch);             //设置发送者邮件地址
if(i=mailadd.Find("\n")!=-1)                             //查找空行
{
ch2[i++]=recvdata[i++];                                  //拷贝字符
GetWindowText(IDC_EDIT1)->SetWindowText(&ch);            //显示邮件内容
         }
}
else
  {
     MessageBox("接收失败！");
              }
         }
}
```

当客户端成功接收到邮件数据后，由于服务器成功执行命令以后会返回 OK，所以客户端需要判断数据中前两个字符是否是 OK，然后在接收的数据最后添加字符“\0”，表示将接收到的字符数据转换为字符串。

字符串转换成功以后，程序调用 CString 类的函数 Find()查找相应的标题字头获取邮件信息。通过在字符串中查找回车符号“\n”，将字符串指针定位到邮件内容处，最后通过循环将邮件内容显示到程序窗口中。

通过自定义函数的分析，用户可以自行在 VC 中进行封装自定义函数，实现自定义功

能。这样对用户学习 C++将会是很好的一次实践机会。

7.5　小　　结

通过本章，用户学习了 SMTP、POP3 命令、一般邮件的数据格式以及服务器响应客户端请求以后返回到客户端的应答内容。本章还讲解了在 VC 中使用资源管理器实现程序界面的美化和编程实现构造命令、发送命令、接收响应数据、分析数据以及显示数据等操作。用户在学习本章的实例程序时，不但可以学习在 VC 中怎样实现程序界面的消息响应，还可以学习邮件客户端与服务器之间的工作原理。

第 8 章　网络文件传输器

网络文件传输是一种基于网络平台的文件操作。通过网络文件传输器可以将需要操作的文件通过网络在两台计算机上实现数据异地传输功能。例如，现在非常流行的 P2P（点对点）传输功能就是通过网络实现用户异地下载或上传文件。在网络中，任何数据都可以进行传输，与网络通信一样，大部分文件传输同样是依靠套接字完成，也可以通过匿名管道等进行传输。

8.1　CFile 类

在 Windows 操作系统下编程操作文件时，可以使用 MFC 类库中的 CFile 类，也可以使用 Win32 API 函数进行编程。对于用户而言，CFile 类比较简单容易使用，所以大部分用户在文件操作编程方面比较偏向于该类。但是使用 API 函数编程可以使用户更加了解程序底层的一些原理。

8.1.1　构造函数

在 MFC 中，关于文件操作的类有很多。其中，最为常用的一个是 CFile 类，这个函数几乎涵盖了所有的文件操作功能。首先，CFile 类的构造函数原型如下：

```
CFile::CFile();                                                //无参数的构造函数
CFile::CFile( LPCTSTR lpszFileName, UINT nOpenFlags );//有参数的构造函数
```

其中，第一个构造函数没有参数，表示在生成文件对象时才调用，此时该对象并未绑定任何文件。如果用户希望构造文件的同时绑定指定文件，那么生成该文件对象以后，需要调用函数 CFile::Open()打开指定文件即可。在 MFC 中，函数 Open()的原型如下：

```
virtual BOOL Open( LPCTSTR lpszFileName, UINT nOpenFlags, CFileException*
pError = NULL );
```

该函数的作用是打开指定文件，并且将该文件与一个文件对象相关联。参数如下：

- 参数 lpszFileName 表示打开的文件名称，该名称可以是一个文件的相对路径或者是绝对路径（表示完整路径）。
- 参数 nOpenFlags 表示将以何种方式打开文件。文件打开方式如表 8.1 所示。
- 参数 pError 表示打开文件时，所发生的异常情况。默认值为 NULL。

例如，用户调用没有参数的构造函数创建文件对象，并且需要将该对象与指定文件绑定在一起再打开文件。代码如下：

```
CFile file;
file.Open("C:\例子.txt",CFile:modeReadWrite);
```

如果用户使用带有参数的构造函数创建文件对象，表示在对象创建的同时，已经与指定文件相关联了。函数中参数意义如下：

- 参数 lpszFileName 表示需要操作的文件路径，该路径可以是绝对路径，也可以是相对路径。例如，打开路径为“C:\例子.txt”的文件，可以将路径直接指定为绝对路径 C:\例子.txt。如果路径指定为.\\例子.txt，那么程序将会在其所在目录下查找该文件。若文件不存在，则会报错。
- 参数 nOpenFlagss 指定文件的打开方式，如表 8.1 所示。

表 8.1　文件打开方式

打 开 方 式	意　　义
CFile::modeCreate	创建新文件并覆盖原有文件
CFile:: modeCreate|CFile::modeNoTruncate	创建文件但不覆盖原有文件
CFile::modeRead	以只读方式打开文件
CFile::modeWrite	以只写方式打开文件
CFile::modeReadWrite	以可读写方式打开文件
CFile::ShareDenyNone	允许其他进程读写文件
CFile::ShareDenyRead	不允许其他进程读文件
CFile::ShareDenyWrite	不允许其他进程写文件
CFile::ShareExclusive	不允许其他进程读写文件

用户在代码中可以调用带有参数的构造函数创建文件对象，并且将文件的打开方式指定为可读可写。代码如下：

```
...                                                //省略部分代码
CFile file('C:\例子.txt',CFile:modeReadWrite);     //创建文件对象
```

用户通过上面的代码，可以创建一个文件对象，并与指定文件相关联，为其设置了打开方式为读写“CFile::modeReadWrite”。

对于用户而言，以上两种构造函数在使用上均可以达到目的。只是在打开文件时，前者需要显式地调用函数 Open()打开文件，而后者则在文件对象创建的同时打开文件，属于隐式。

8.1.2　读写文件

当用户创建文件对象成功以后，可以调用相关的操作函数对其进行读写操作。在 MFC 中，进行文件读写操作的函数分别是 CFile 类的函数 Read()和 Write()。原型分别如下：

```
virtual UINT Read(void* lpBuf, UINT nCount);              //读文件
virtual void Write(const void* lpBuf, UINT nCount);       //写文件
```

两个函数的参数及其意义均相同，如下所示：

- 参数 lpBuf 表示指向缓冲区的指针。

- 参数 nCount 表示需要操作的字节数。

其中，读文件的函数 Read()如果调用成功，则会返回实际读取到的字节数目。用户在程序中使用这两个函数对文件进行操作，代码如下：

```
...                                                  //省略部分代码
char *text[100];                                     //定义字符数组
CFile file('C:\例子.txt',CFile:modeReadWrite);       //创建文件对象
file.Read(text,100);                                 //将文件数据读取到指定缓冲区中
file.Write(text,100);                                //将缓冲区中的数据写到文件中
...                                                  //省略部分代码
```

上述代码中，创建文件对象以后，分别调用函数 Read()和 Write()对该文件进行读写操作。如果文件中原有数据为空或者不足用户指定的数目时，函数 Read()将返回实际读取到的字节数。代码如下：

```
...                                                  //省略部分代码
int n=0;                                             //定义并初始化变量
CString str;                                         //定义字符串
n=file.Read(text,100);                               //将文件数据读取到指定缓冲区中
if(n==0)                                             //文件为空
{
  MessageBox("文件为空！");
  }
else
{
  str.Format("实际读取到的文件字节数为 d%\n",n);     //格式化字符串
MessageBox(str);
}
```

上述的代码实现了读取文件，并且根据读取到的文件数据数目判断文件是否为空。若为空则提示用户原有文件数据为空，否则显示实际读取到的文件数据数目。

8.1.3　文件关闭

用户操作完文件后，需要将文件关闭，否则将发生错误或者前面的操作失败。实现文件关闭操作的函数分别是函数 Abort()和 Close()，原型分别如下：

```
virtual void Abort();            //强制关闭文件并销毁文件对象
virtual void Close();            //正常关闭文件
```

以上两个函数的作用都是关闭文件。但是，使用前者关闭文件是在操作文件发生异常时才使用该函数对文件实行强制关闭，如果文件属于正常关闭时使用后者即可。

注意： 一般情况下，在 Windows 操作系统中操作文件，例如写入文件，系统均提供缓冲机制，即在文件正常关闭才将数据写入文件所在的物理盘符中。

当用户希望操作文件时，为了避免数据丢失，需要将数据立刻写入文件中，则可以使用函数 CFile::Flush()。该函数原型如下：

```
virtual void Flush();
```

该函数将数据强制写入文件中，避免数据丢失。用户使用该函数强制写入数据后，关闭文件，代码如下：

```
...                                              //省略部分代码
char *text[3]={a,b,c};                           //定义并初始化字符数组
CFile file("C:\例子.txt",CFile:modeReadWrite);  //创建文件对象
file.Write(text,3);                              //将缓冲区中的数据写到文件中
file.Flush();                                    //强制写入数据
file.Close();                                    //正常关闭文件
//file.Abort();                                  //强制关闭文件
...                                              //省略部分代码
```

在代码中，用户首先定义并初始化了一个字符数组指针变量，然后创建文件对象 file，再强制将字符数组中的数据写入与该文件对象相关联的文件中。

8.1.4　文件定位

通常情况下，用户在操作某一文件时，希望从文件的某一特定处开始读取或写入文件。这时，用户将使用定位文件的函数，如 CFile 类的函数 Seek()。函数原型如下：

```
virtual LONG Seek(LONG loff, UINT nFrom);
virtual DWORD GetPosition()const;
```

函数 Seek 用于随机访问文件中的数据，参数意义如下：

- 参数 loff 表示确定指针移动的字节数，正值表示指针向后移动，负值表示指针向前移动。
- 参数 nFrom 表示指针移动的模式，其取值如表 8.2 所示。

表 8.2　文件指针移动模式取值

取　　值	意　　义
CFile::begin	从文件开头向后移动 loff 个字节
CFile::Current	从文件当前位置向后移动
CFile::end	从文件结尾向前移动，此时 loff 必须为负值，表示向前移动

该函数如果调用成功，返回值为新的相对于文件开头的字节偏移量。当文件第一次打开时，文件指针均在文件开始处。例如，用户打开文件，并在文件结尾处添加几个字符。代码如下：

```
...                                              //省略部分代码
char *text[3]={a,b,c};                           //定义并初始化字符数组
CFile file("C:\例子.txt",CFile:modeReadWrite);  //创建文件对象
file.Seek(-1,CFile::end);                        //将文件指针定位到文件结尾
file.Write(text,3);                              //将缓冲区中的数据写到文件中
file.Flush();                                    //强制写入数据
file.Close();                                    //正常关闭文件
...                                              //省略部分代码
```

在代码中，用户使用函数 Seek()将文件指针定位到文件结尾处，然后使用 Write()函数将字符写入文件。

注意： 用户将文件指针定位到文件结尾处时，一定要将参数 loff 指定为负数，表示文件指针向前移动。否则，程序将报错。

在 MFC 中，还有一个函数 GetPosition()用于获取当前文件的文件指针位置。该函数原型如下：

```
virtual DWORD GetPosition( )const;
```

该函数如果调用成功，将返回相对于文件开头位置的文件指针字节偏移量。用户可以使用该函数获取当前文件指针的位置，再在该文件指针处写入或者读取数据。代码如下：

```
...                                                //省略部分代码
char *text[3]={a,b,c};                             //定义并初始化字符数组
CFile file("C:\例子.txt",CFile:modeReadWrite);     //创建文件对象
file. GetPosition();                               //将文件指针定位到文件结尾
file.Write(text,3);                                //将缓冲区中的数据写到文件中
//file.Read(text,3);                               //读取当前文件指针后的 3 个数据
file.Flush();                                      //强制写入数据
file.Close();                                      //正常关闭文件
...                                                //省略部分代码
```

在本节中，主要向用户介绍了在 MFC 中 CFile 类的主要函数的原型以及用法。关于该类的其他函数方法将在后面的实例中讲述。

8.2　使用 API 函数操作文件

当用户使用 MFC 编程时，除了使用 CFile 类操作文件以外，还可以使用 API 函数（应用程序接口函数）中有关文件操作的函数进行编程。使用 API 函数编程可以让用户更加了解文件操作编程的原理以及方法。

8.2.1　创建文件

在 API 函数中，用户可以使用函数 Create()进行创建文件对象。该函数原型如下：

```
HANDLE CreateFile(
  LPCTSTR lpFileName,
  DWORD dwDesiredAccess,
  DWORD dwShareMode,
  LPSECURITY_ATTRIBUTES lpSecurityAttributes,
  DWORD dwCreationDisposition,
  DWORD dwFlagsAndAttributes,
  HANDLE hTemplateFile
);
```

如果该函数调用成功，则返回所创建的文件对象句柄。用户可以使用该对象句柄对文件进行操作。其参数意义如下：

- 参数 lpFileName 表示文件名。该文件名可以包含指定路径，表示将在指定路径上创建该文件，否则，函数将在工程目录下创建该文件。

- 参数 dwDesiredAccess 表示文件的存取方式。存取方式如表 8.3 所示。

表 8.3　文件存取方式

取　　值	意　　义
0	表示以默认方式对文件进行操作
GENERIC_READ	表示以只读方式操作文件
GENERIC_WRITE	表示以只写方式操作文件

- 参数 dwShareMode 用于指定文件的共享模式。即当程序打开该文件后是否允许其他进程或程序以同种方式再次打开该文件。其取值如表 8.4 所示。

表 8.4　文件共享模式取值

取　　值	意　　义
0	不再允许其他程序再次打开该文件
FILE_SHARE_DELETE	允许其他程序对该文件进行删除操作
FILE_SHARE_READ	允许其他代码以只读方式打开该文件
FILE_SHARE_WRITE	允许其他代码以只写方式打开该文件

- 参数 lpSecurityAttributes 是指向结构体 SECURITY_ATTRIBUTES 的指针。用来为创建的文件指定安全属性。一般情况下，均将该参数指定为 NULL，表示默认的安全属性。
- 参数 dwCreationDisposition 表示函数是打开文件还是创建文件。其取值如表 8.5 所示。

表 8.5　文件创建方式

取　　值	意　　义
CREATE_NEW	创建新文件，如果文件已经存在，则函数执行失败
CREATE_ALWAYS	创建新文件，如果文件已经存在，则函数会覆盖原文件并清除其所存在的所有文件属性
OPEN_EXISTING	打开已存在的文件，如文件不存在，则函数执行失败
OPEN_ALWAYS	如果文件不存在，则创建新文件，否则打开该文件
TRUNCATE_EXISTING	打开已经存在的文件并将其内容清空，如果不存在，则函数执行失败

- 参数 dwFlagsAndAttributes 指定文件的新属性。其取值如表 8.6 所示。

表 8.6　文件属性值

取　　值	意　　义
FILE_ATTRIBUTE_ARCHIVE	标记存档属性
FILE_ATTRIBUTE_HIDDEN	标记隐藏属性
FILE_ATTRIBUTE_READONLY	标记只读属性
FILE_ATTRIBUTE_SYSTEM	标记为系统文件属性
FILE_ATTRIBUTE_TEMPORARY	指定临时文件属性

注意：如果将文件的属性指定为 FILE_ATTRIBUTE_TEMPORARY，则函数会将文件的属性指定为临时文件。操作系统会将临时文件的内容保存在内存中以便程序加快文件的存取速度，但当程序使用完成后，系统会将其删除，同时还可以为临时文件指定操作方式。临时文件的部分操作方式如表 8.7 所示。

表 8.7　临时文件的操作方式

操 作 方 式	意　　义
FILE_FLAG_DELETE_ON_CLOSE	当程序关闭后，系统会立即删除该临时文件
FILE_FLAG_OVERLAPPED	设置异步读写该临时文件
FILE_FLAG_WRITE_THROUGH	系统将不会对该临时文件使用缓存，不论文件有任何修改都将被立刻写入硬盘中

- 参数 hTemplateFile 指定文件模板的句柄，设置该参数后，系统会复制该文件模板的所有属性到当前所创建的文件中，用户可以将该参数设置为 NULL。

例如，用户使用函数 CreateFile()创建一个新文件。代码如下：

```
...                                                  //省略部分代码
HANDLE handle;                                       //定义文件句柄
handle=::CreateFile("C:\例子.txt", 0, FILE_SHARE_DELETE| FILE_SHARE_READ|
FILE_SHARE_WRITE, NULL, CREATE_ALWAYS, FILE_ATTRIBUTE_ARCHIVE|
  FILE_ATTRIBUTE_SYSTEM,NULL);                       //创建文件
if(handle!=-1)
{
  MessageBox("文件创建成功！");
   }
else
{
  MessageBox("文件创建失败！");
  }
```

上述代码将在 C 盘下创建一个名称为“例子”的文本文件。如果创建成功，则函数返回新创建文件对象的句柄，否则，函数将返回–1。

注意：用户在调用 API 函数时。需要在该函数前使用符号“::”进行调用，表示调用的函数为 Win32 API 函数。否则，程序将调用 MFC 中相应函数。

8.2.2　操作文件

用户使用 API 函数进行文件编程时，读取文件的操作函数是 ReadFile()，该函数原型如下：

```
BOOL ReadFile(
  HANDLE hFile,
  LPVOID lpBuffer,
  DWORD nNumberOfBytesToRead,
  LPDWORD lpNumberOfBytesRead,
  LPOVERLAPPED lpOverlapped
);
```

该函数的作用是从指定文件中读取相应大小的数据到指定的缓冲区中。如果函数调用失败，则返回 0。否则返回非 0 值。其参数含义如下：

- ❑ 参数 hFile 表示将要操作的文件对象句柄，即使用函数 CreateFile()成功创建文件后返回的文件句柄。
- ❑ 参数 lpBuffer 是一个指向缓冲区的指针，函数读取到的数据将被存放到该缓冲区中。
- ❑ 参数 nNumberOfBytesToRead 表示用户将要读取的字节数目。
- ❑ 参数 lpNumberOfBytesRead 是一个指向 DWORD 类型的指针变量，用于返回实际读取的字节数目。
- ❑ 参数 lpOverlapped 是指向结构体 overlapped 的指针。一般情况下，用户将该参数设置为 NULL 即可。

编程时，与读取文件的函数 ReadFile()相对应的函数是 WriteFile()，该函数的作用是写入数据到指定文件中。该函数原型如下：

```
BOOL WriteFile(
  HANDLE hFile,
  LPCVOID lpBuffer,
  DWORD nNumberOfBytesToWrite,
  LPDWORD lpNumberOfBytesWritten,
  LPOVERLAPPED lpOverlapped
);
```

如果该函数成功调用，则返回 0，否则返回非 0 值。其参数的意义与函数 ReadFile()的参数意义是一样的。

注意：当使用该函数写入数据到文件时，被写入的数据通常会被操作系统暂时保存在一个缓冲区中，等到文件关闭或数据大小与缓冲区大小一样时，才会被系统一并写入文件所在的物理磁盘中。

例如，用户使用 API 函数对创建的文件进行读写操作。代码如下：

```
...                                              //省略部分代码
HANDLE handle;                                   //定义文件句柄
char buffer[100];                                //定义缓冲区
int i;                                           //接收实际操作的字节数
CString str;                                     //定义字符串变量
handle=::CreateFile("C:\例子.txt", 0, FILE_SHARE_DELETE| FILE_SHARE_READ|
FILE_SHARE_WRITE, NULL, CREATE_ALWAYS, FILE_ATTRIBUTE_ARCHIVE|
  FILE_ATTRIBUTE_SYSTEM,NULL);                   //创建文件
if(handle==-1)                                   //判断文件是否创建成功
{
  MessageBox("文件创建失败！");
   }
else
{
 if(::ReadFile(handle,&buffer,100,i,NULL))  //读取文件数据到指定缓冲区中
{
  str.Format("实际读取到 d%\n",i);               //格式化字符串
MessageBox(str);
::WriteFile(handle,str.GetBuff(1),sizeof(str),i,NULL);
```

```
                                                    //将字符串写入文件中
    }
else
{
  MessageBox("读取文件失败！");
   }
  }
```

在上面的代码中，只有当程序关闭时，系统才会将字符串数据写入文件所在的物理磁盘中。如果用户希望数据被立刻写入文件所在的磁盘中时，可以使用函数 FlushFileBuffers()将数据强制写入文件中。该函数原型如下：

```
BOOL FlushFileBuffers(HANDLE hFile);
```

该函数的唯一参数 hFile 表示被操作文件的对象句柄。例如，将上面示例程序中的数据立刻写入文件，代码如下：

```
...                                          //省略部分代码
 if(::ReadFile(handle,&buffer,100,i,NULL)) //读取文件数据到指定缓冲区中
{
  str.Format("实际读取到 d%\n",i);           //格式化字符串
MessageBox(str);
::WriteFile(handle,str.GetBuff(1),sizeof(str),i,NULL);
                                             //将字符串写入文件中
:: FlushFileBuffers(handle);                 //强制向文件中写入数据
    }
...                                          //省略部分代码
```

通过以上代码，程序会将缓冲区中的数据立刻写入指定文件中。这样做，可以避免数据的丢失，加强了数据的安全性。

用户关闭文件的操作可以通过 API 函数 CloseHandle()，该函数可以关闭任何对象的句柄。其原型如下：

```
BOOL CloseHandle(HANDLE hObject );
```

该函数只有一个参数，即需要关闭的对象句柄。例如，用户操作完文件后，需要关闭该文件，使用该函数进行文件的关闭操作。代码如下：

```
...                                          //省略部分代码
handle=::CreateFile("C:\例子.txt", 0, FILE_SHARE_DELETE| FILE_SHARE_READ|
FILE_SHARE_WRITE, NULL, CREATE_ALWAYS, FILE_ATTRIBUTE_ARCHIVE|
  FILE_ATTRIBUTE_SYSTEM,NULL);               //创建文件
...                                          //省略部分代码
::CloseHandle(handle);                       //关闭对象句柄
...                                          //省略部分代码
```

如果用户使用的文件创建函数是 MFC 中库函数，同样可以使用函数 CloseHandle()进行关闭。因为在每个对象中均包含了一个表示该对象句柄的变量 m_hWnd。例如，用户使用 CFile 类创建文件，然后使用该函数进行关闭，代码如下：

```
CFile file("C:\例子.txt",CFile:modeReadWrite);       //创建文件对象
...                                                  //省略部分代码
::CloseHandle(file.m_hWnd);                          //关闭对象句柄
```

通过本节的学习，用户应该了解并能够使用 API 函数进行基本的文件操作编程。由于本章中的实例程序只涉及到一些基本的 API 函数，所以，在这里不再赘述。如果用户愿意继续学习其他作用的 API 函数，可以翻阅一些关于 API 讲解的参考书。

8.3　内存映射文件

内存映射文件是与虚拟内存相似的一种内存地址空间，只有在程序需要使用时才会将该空间中的内容提交给物理磁盘。使用内存映射文件不但能减少读取和写入文件的时间，还可以避免对文件的多次输入输出操作和为文件操作频繁地申请内存缓冲区。

1. 相关函数

用户在实际编程时，可以使用 API 函数 CreateFileMapping()打开或者创建内存映射文件对象。该函数原型如下：

```
HANDLE CreateFileMapping(
  HANDLE hFile,
  LPSECURITY ATTRIBUTES lpFileMappingAttributes,
  DWORD flProtect,
  DWORD dwMaximumSizeHigh,
  DWORD dwMaximumSizeLow,
  LPCTSTR lpName
);
```

该函数如果调用成功，则返回新创建的内存映射文件句柄。否则，将返回 0。其中，参数意义如下：

- 参数 hFile 表示需要映射的文件对象句柄。如果该文件对象句柄已经存在，那么函数将建立该文件的内存映射对象。否则，函数将建立一个共享内存。
- 参数 lpFileMappingAttributes 将指定该内存映射文件的安全属性，在这里设置为 NULL。
- 参数 flProtect 指定内存映射文件的保护类型，其取值如表 8.8 所示。

表 8.8　内存映射文件的保护类型

取　　值	意　　义
PAGE_READONLY	设置该内存映射文件为只读
PAGE_READWRITE	设置该内存映射文件为可读写

- 参数 dwMaximumSizeHigh 和 dwMaximumSizeLow 共同指定该内存映射文件的长度，若函数创建共享内存，则需要为这两个参数指定值。否则，将两个参数均设置为 0，表示创建的内存映射文件长度与磁盘上已经存在的文件长度一样。
- 参数 lpName 表示创建的内存映射文件名。

当该内存映射文件创建成功以后，其他进程或者程序可以调用函数 OpenFileMapping()根据内存映射文件名打开已经存在的内存映射文件。函数 OpenFileMapping()原型如下：

```
HANDLE OpenFileMapping(
  DWORD dwDesiredAccess,
```

```
  BOOL bInheritHandle,
  LPCTSTR lpName
);
```

该函数调用成功，将返回打开的内存映射文件对象的句柄。否则，返回 0。参数意义如下：

- 参数 dwDesiredAccess 指定打开内存映射文件的保护类型。取值如表 8.9 所示。

表 8.9　内存映射文件的保护类型

取　　值	意　　义
FILE_MAP_WRITE	以可写属性打开该内存映射文件
FILE_MAP_READ	以可读属性打开该内存映射文件

- 参数 bInheritHandle 指定该函数返回的句柄是否可以被继承。
- 参数 lpName 指定将要打开的内存映射文件的文件名。

例如，用户创建并打开文件名为 lymlrl 的内存映射文件，代码如下：

```
...                                                 //省略部分代码
HANDLE handle;
handle= ::CreateFileMapping(0,NULL, PAGE READWRITE,0,0,"lymlrl");
                                                    //创建命名的共享内存
:: OpenFileMapping(FILE MAP WRITE| FILE MAP READ,false,"lymlrl");
                                                    //打开该共享内存
...                                                 //省略部分代码
```

以上代码创建了一个名称为 lymlrl 的共享内存，并在创建后将其打开。当用户打开内存映射文件成功后，需要对该内存映射文件进行存取操作，那么用户需要得到该内存映射文件的内存地址。实现该功能的 API 函数是 MapViewOfFile()，该函数原型如下：

```
LPVOID MapViewOfFile(
  HANDLE hFileMappingObject,              //内存映射文件的对象句柄
  DWORD dwDesiredAccess,                  //指定保护类型
  DWORD dwFileOffsetHigh,                 //从文件的指定地址开始映射
  DWORD dwFileOffsetLow,                  //指定映射停止的文件指针位置
  DWORD dwNumberOfBytesToMap              //需要映射的字节数，若为 0 则映射整个文件
);
```

如果该函数调用成功，则返回内存映射文件的内存地址。否则，将返回 NULL。当用户不再使用该内存映射文件时，应将其对象关闭。实现映射文件对象关闭的函数是 UnmapViewOfFile()。该函数原型如下：

```
BOOL UnmapViewOfFile(
  LPCVOID lpBaseAddress
);
```

该函数调用成功，则返回 true。否则，返回 false。其参数 lpBaseAddress 表示内存映射文件的内存地址。

2. 示例代码

用户通过 8.2 节对内存映射文件的介绍以及相关的编程操作函数的讲解，对内存映射文件的作用和操作已经有了初步的认识。在本节中将通过编写一个内存映射文件实例，向

用户进一步介绍内存映射文件的编程操作。例如，用户向一个共享内存中写入数据后，再关闭该内存映射文件。代码如下：

```
...                                                         //省略部分代码
HANDLE handle;                                              //定义句柄
LPVOID add;                                                 //表示映射地址
char text[3]={a,b,c};                                       //写入文件的数据
handle= ::CreateFileMapping(0,NULL, PAGE_READWRITE,0,0,"lymlrl");
                                                            //创建命名的共享内存
:: OpenFileMapping(FILE_MAP_WRITE| FILE_MAP_READ,false,"lymlrl");
                                                            //打开该共享内存
add=:: MapViewOfFile(handle, FILE_MAP_WRITE,0,0,0);//映射文件
strcpy(add,&text);                                          //复制数据到映射文件
 :: UnmapViewOfFile(add);                                   //撤销映射
```

用户从代码中可以看到，使用内存映射文件进行数据输入输出时可以使用数据复制函数 strcpy()实现数据的保存，减少程序对文件的访问。

注意：*在本章实例中，将会使用内存映射文件对文件数据进行读取和保存。所以，用户应该重视本节关于内存映射文件的相关内容。*

8.4　使用 Socket 传输文件

本节将讲述本章的核心功能，使用 Socket（套接字）编程实现文件数据的传输。其基本过程是，程序在发送文件时，首先将本地文件数据读取到指定缓冲区后，再使用套接字将缓冲区内容发送到远程计算机上。当程序接收文件时，首先将数据接收并存入缓冲区中，然后创建相应文件并将缓冲区内容写入文件即可。

8.4.1　创建套接字

本节中创建套接字的方法与前面章节中创建套接字的方法一样，所以，本节不再赘述。客户端创建套接字，代码如下：

```
...                                             //省略部分代码
SOCKET s;                                       //定义套接字对象
sockaddr_in addr;                               //定义网络地址结构对象
addr.sin_family=AF_INET;                        //为地址结构中的成员赋值
addr.sin_port=htons(100);                       //使用 100 号端口
addr.sin_addr.S_un.S_addr=inet_addr(smtpip);    //转换对方计算机的 IP 地址
s=::socket(AF_INET,SOCK_STREAM,IPPROTO_TCP);    //创建套接字
    ...                                         //省略部分代码
```

如果是服务器端，则创建套接字之后还需要将套接字与本地地址相绑定，代码如下：

```
...                                             //省略部分代码
SOCKET s;                                       //定义套接字对象
sockaddr_in addr;                               //定义网络地址结构对象
```

```
addr.sin_family=AF_INET;                          //为地址结构中的成员赋值
addr.sin_port=htons(100);
addr.sin_addr.S_un.S_addr=inet_addr(127.0.0.1);
                                                  //假设服务器地址为 127.0.0.1
s=::socket(AF_INET,SOCK_STREAM,IPPROTO_TCP);         //创建套接字
::bind(s,(sockaddr *)addr,sizeof(addr));             //绑定套接字
:: listen(s,3);                                      //监听套接字
   ...                                               //省略部分代码
```

对于客户端而言，套接字创建成功以后，便可以连接服务器。代码如下：

```
...                                                  //省略部分代码
s=::socket(AF_INET,SOCK_STREAM,IPPROTO_TCP);         //创建套接字
::connect(s, (sockaddr *)addr, sizeof(addr));        //连接服务器
...                                                  //省略部分代码
```

对于服务器而言，当程序在指定端口上发现有客户端发送的连接请求到来时，则调用函数 accept()对该连接请求进行应答，并返回一个新的套接字句柄。代码如下：

```
SOCKET s,s1;                                         //定义套接字对象
sockaddr_in addr2;                                   //定义网络地址结构对象
...                                                  //省略部分代码
s=::socket(AF_INET,SOCK_STREAM,IPPROTO_TCP);         //创建套接字
::bind(s,(sockaddr *)addr,sizeof(addr));             //绑定套接字
:: listen(s,3);                                      //监听套接字
s1=::accept(s,( sockaddr *)addr2,sizeof(addr2);      //应答连接请求
   ...                                               //省略部分代码
```

用户可以从上面的代码中看到，如果函数 accept()调用成功，则返回一个新的套接字句柄并且获得与其连接的客户端地址信息。服务器与客户端的数据需要依靠函数 accept()返回的新套接字句柄进行传输。

8.4.2　关闭套接字

当用户不再需要使用已经创建的套接字以后，可以调用 API 函数 CloseHandle()对其进行关闭。该函数原型如下：

```
BOOL CloseHandle(HANDLE hObject );
```

该函数执行成功，则返回 true。否则，返回 false。参数 hObject 表示需要关闭的对象句柄。例如，用户关闭上面代码中创建的套接字句柄 s，代码如下：

```
...                                               //省略部分代码
SOCKET s;                                         //定义套接字对象
sockaddr_in addr;                                 //定义网络地址结构对象
addr.sin_family=AF_INET;                          //为地址结构中的成员赋值
    addr.sin_port=htons(100);                     //使用端口号 100
    addr.sin_addr.S_un.S_addr=inet_addr(smtpip);     //转换对方计算机的 IP 地址
    s=::socket(AF_INET,SOCK_STREAM,IPPROTO_TCP);     //创建套接字
    ::CloseHandle(s);                                //关闭套接字
```

如果用户在代码中使用 MFC 中创建套接字的类或者方法创建套接字句柄 s，则使用

CloseHandle()函数进行关闭时，应该将参数形式修改为“对象.m_hWnd”。

8.4.3　发送文件

当客户端连接服务器成功以后，客户端便可以向服务器传输文件数据了。同样，对于服务器而言，也可以通过 API 函数 accept()返回的新套接字句柄向客户端传输或接收文件数据。由于在本实例中，服务器和客户端都具有发送和接收文件数据的功能，所以在本节中只针对客户端发送文件功能进行讲解。代码如下：

```
...                                               //省略部分代码
SOCKET s;                                         //定义套接字对象
char text[100]                                    //定义缓冲区
CFile file("C:\例子.txt",CFile:modeReadWrite);  //创建文件对象
CString str;                                      //定义字符串
sockaddr_in addr;                                 //定义网络地址结构对象

addr.sin_family=AF_INET;                          //为地址结构中的成员赋值
    addr.sin_port=htons(100);                     //使用端口号 100
    addr.sin_addr.S_un.S_addr=inet_addr(smtpip);//转换服务器 IP 地址
    s=::socket(AF_INET,SOCK_STREAM,IPPROTO_TCP);//创建套接字
   if(::connect(s, (sockaddr *)addr, sizeof(addr))!=-1)
                                                  //连接服务器
     {
       str=file.GetFileTitle();                   //获取文件名“例子.txt"
       ::send(s,str.GetBuffer(1),sizeof(str)); //发送文件名到服务器
       file.Read(text,100);                       //读取文件
       while(text!=EOF)                           //判断文件是否结束
      {
file.Read(text,100);                              //读取 100 字节的文件数据
       ::send(s,&text,100);                       //发送文件数据到服务器
          }
::CloseHandle(file.m_hWnd);                       //关闭对象句柄
:: CloseHandle(s);
}
```

在代码中，用户创建一个文件对象 file，然后调用函数 GetFileTitle()获取该文件的文件标题（包括文件的后缀名）。通过函数 send()将获取的文件标题发送到服务器，以便服务器创建相同文件名的文件。最后，程序循环读取文件内容到缓冲区 text 中，并且将每次读取到的数据发送到服务器。

🔔**注意：** 在程序中，是以字符 EOF 标识文件结束。

8.4.4　接收文件

当客户端将文件数据发送到服务器以后，服务器应该负责接收文件并且在本地磁盘中创建相应文件接收数据。接收时，服务器第一次接收到的数据应该为文件名，后面接收到的数据均是文件数据。代码如下：

```
...                                                     //省略部分代码
SOCKET s,s1;                                            //定义套接字对象
sockaddr_in addr2;                                      //定义网络地址结构对象
char text[100]={0};                                     //定义缓冲区
s=::socket(AF_INET,SOCK_STREAM,IPPROTO_TCP);            //创建套接字
::bind(s,(sockaddr *)addr,sizeof(addr));                //绑定套接字
:: listen(s,3);                                         //监听套接字
s1=::accept(s,( sockaddr *)addr2,sizeof(addr2);         //应答连接请求
if(s1!=NULL)
{
  ::recv(s1,&text,100);                                 //接收文件名
  if(text!=0)
  {
    CFile file((LPSTR)text, CFile:modeReadWrite);       //根据文件名创建文件
    if(file!=NULL)                                      //判断文件是否创建成功
    {
      while(text!=0)                                    //循环接收数据
        {
                ::recv(s1,&text,100);                   //接收数据
file.Write(&text,100);                                  //写入文件
}
file.Close();                                           //关闭文件或套接字句柄
::CloseHandle(s);
::CloseHandle(s1);
     }
   }
}
```

以上程序表示服务器成功应答客户端的连接请求后，可以通过返回的新套接字与客户端进行数据传输。首先，服务器接收文件名，如果接收的文件名不为 NULL，则创建相同名称的文件。然后，程序循环接收客户端发送的文件数据，并将这些数据写入新创建的文件中。

注意：在服务器接收实例程序中，本节使用了文件结束符 EOF 或者是 0 来表示文件结束。如果用户有其他方法在客户端和服务器之间约定文件的结束标志，那么用户可以自行尝试将以上代码进行修改。调试程序，查看自定义的约定方式是否成立。

8.5　服务器代码

通过前面的学习，用户已经对文件在客户端和服务器之间进行传输的基本原理以及方法有了了解。在本节中，将通过编写服务器代码向用户详细讲解服务器接收文件数据和发送文件数据功能的具体实现方法以及实例代码。

8.5.1　服务器功能

在本章实例中，服务器应当具有向客户端发送文件以及接收客户端文件数据的功能。当服务器接收到客户端的文件数据以后，首先在本地目录中创建一个与接收文件名相同的

文件。当文件创建成功以后，通过循环方式接收客户端发送的文件数据，并将这些文件数据写入新文件中，然后关闭文件。

在服务器中还需要实现一个功能，即与客户端用户进行交流的平台。这个功能与第 6 章中所讲述的网络通信器的功能一样能够互相发送消息。因此，在本节中关于这个功能的实现原理就不再向用户进行讲述。如果用户对该功能的实现方法等还不熟悉，请复习第 6 章中的内容。

8.5.2　创建服务器对话框

服务器端功能的实现代码将在 VC 环境下进行编写和调试。首先，用户需要定义服务器端界面是基于对话框模式。

1. 创建工程

用户使用 VC 创建基于 MFC 应用程序的服务器工程，可以使用 VC 的工程向导进行工程创建。其创建步骤如下。

（1）新建一个工程，工程属性为 MFC AppWizard[exe]，工程名称修改为“服务器”，如图 8.1 所示。

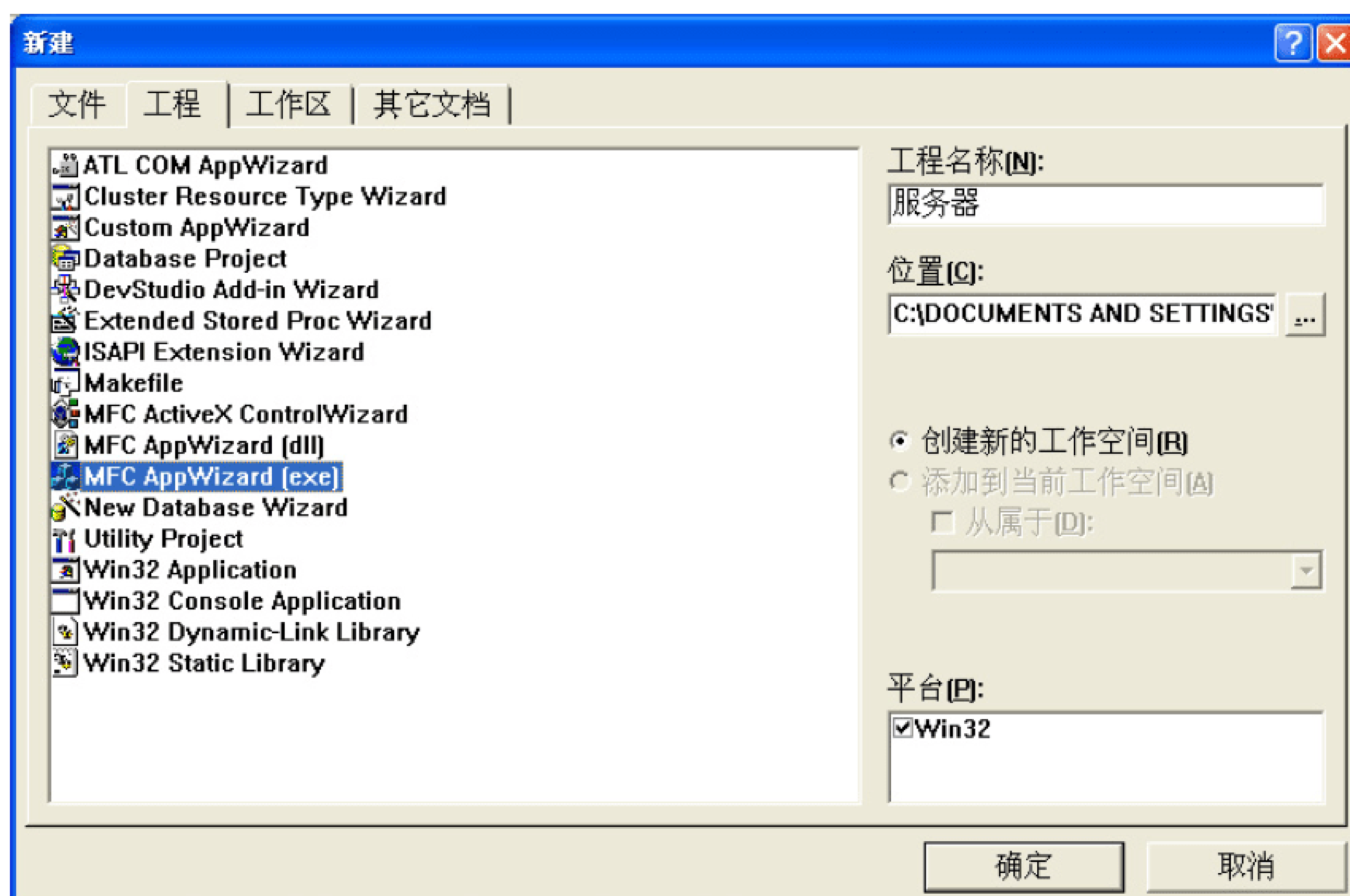


图 8.1　新建工程对话框

（2）单击“确定”按钮，进入应用程序向导步骤 1，选择应用程序类型，如图 8.2 所示。

（3）单击“下一步”按钮，进入应用程序向导步骤 2，选择应用程序需要 Windows Sockets 的支持，如图 8.3 所示。

（4）单击“完成”按钮，完成服务器工程的创建。

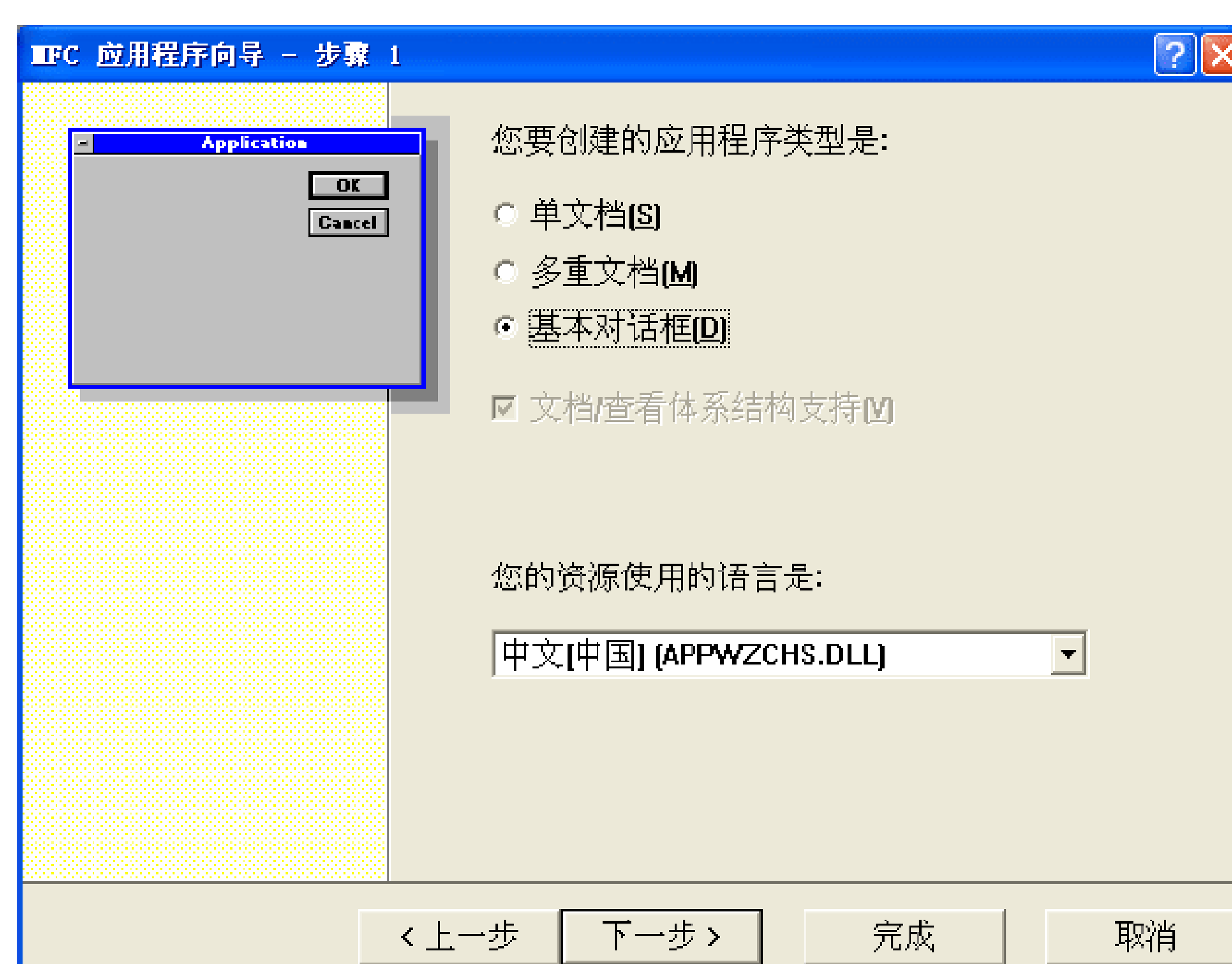


图 8.2　选择应用程序类型为对话框模式



图 8.3　选择应用程序支持套接字功能

2. 设计界面

用户需要根据 8.2.1 节中所示的服务器功能进行服务器界面设计。首先，打开工程资源管理器中的对话框资源，如图 8.4 所示。

注意： *在图 8.4 中所示对话框界面是在 VC 资源管理器中自动生成的对话框界面。在实际编程中，需要用户向该界面中拖动各种控件以满足其功能。*

然后，用户拖动控件到对话框面板上相应位置，并调整其大小。服务器界面的设计效果如图 8.5 所示。

用户在对话框界面中，总共放置了 7 个控件。关于这些控件的 ID、属性以及作用如表 8.10 所示。

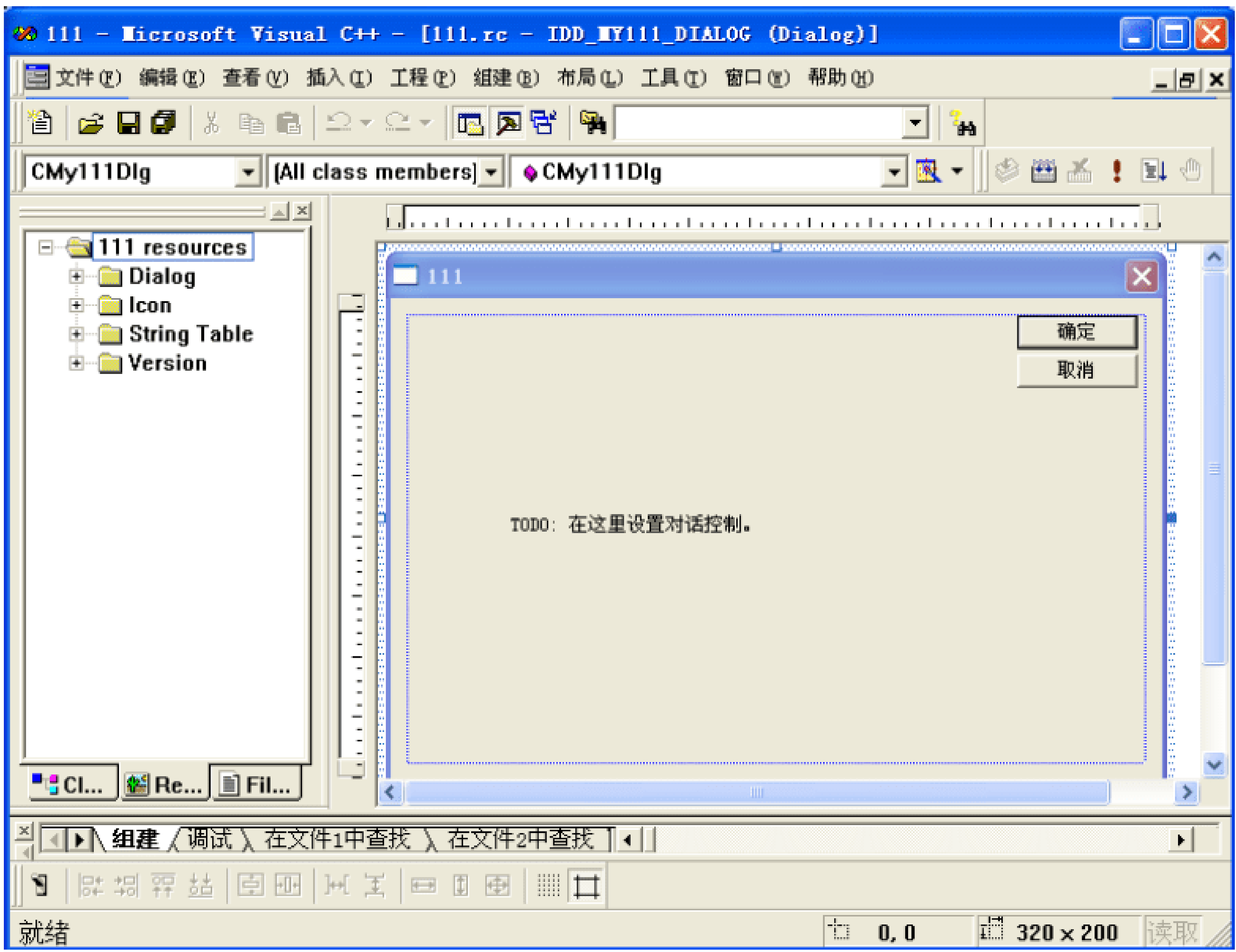


图 8.4　打开对话框初始界面



图 8.5　服务器界面效果

表 8.10　控件ID以及属性

控件 ID	控 件 属 性	控 件 作 用
IDC_SAVE	按钮	保存接收到的文件
IDC_LIULAN	按钮	发送文件的位置
IDC_CLEAR	按钮	清空信息
IDC_EDIT1	编辑框	显示服务器和客户端之间聊天信息
IDC_STATIC	静态控件	标识作用
IDC_EDIT2	编辑框	服务器将发送的消息
IDC_SEND	按钮	发送消息

本节向用户介绍了服务器工程的创建方法以及服务器界面设计所需的控件类型以及作用等，基本完成了服务器界面的设计步骤。但是，要让该服务器对话框实现原定的功能，还需要用户实现每一个控件的响应函数。对于实现控件的响应函数将在 8.5.3 节中讲解。

8.5.3　程序初始化

在本节中，将通过界面初始化函数向用户介绍该界面中各个控件的显示状态以及服务器端套接字的初始化等。

1．控件初始化状态

当界面初始化时，用户应该使界面中部分控件处于禁用状态。所以，用户需要在窗口初始化函数中编写相关程序实现控件禁用功能。代码如下：

```
BOOL CMyDlg::OnInitDialog()
{
    CDialog::OnInitDialog();
    ASSERT((IDM_ABOUTBOX & 0xFFF0) == IDM_ABOUTBOX);
    ASSERT(IDM_ABOUTBOX < 0xF000);
    CMenu* pSysMenu = GetSystemMenu(FALSE);
    ...                                                  //省略部分代码
    SetIcon(m_hIcon, TRUE);
    SetIcon(m_hIcon, FALSE);
    GetDlgItem(IDC_EDIT1)->EnableWindow(false);          //禁用信息显示窗口
GetDlgItem(IDC_SAVE)->EnableWindow(false);               //禁用保存文件按钮
GetDlgItem(IDC_LIULAN)->EnableWindow(false);             //禁用发送文件按钮
GetDlgItem(IDC_CLEAR)->EnableWindow(false);              //禁用清除信息按钮
...                                                      //省略部分代码
    return TRUE;
}
```

将以上代码在 VC 编译器中进行编译、运行。程序窗口初始化界面，如图 8.6 所示。

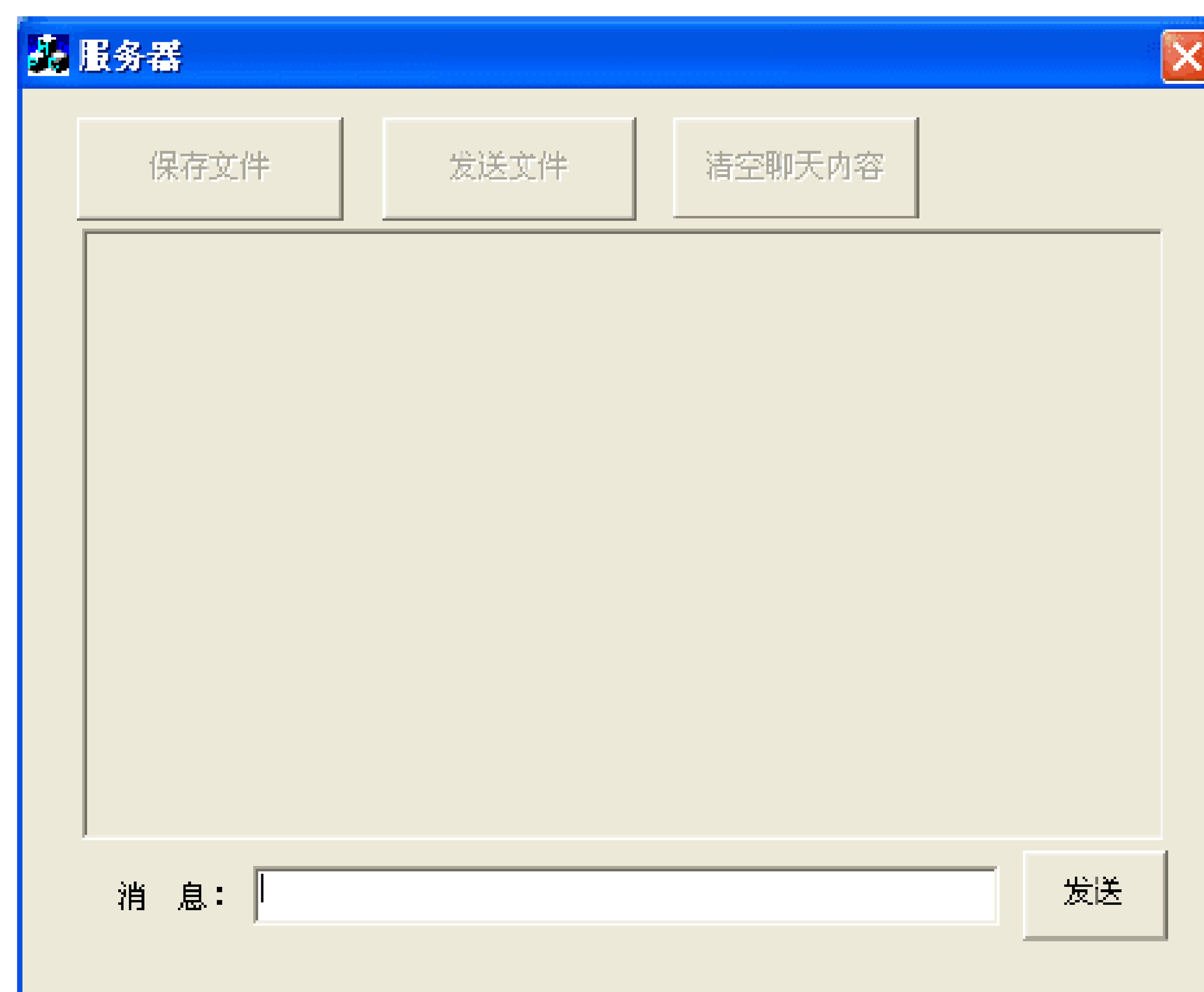


图 8.6　窗口初始化效果

注意：程序窗口运行时，主要的功能按钮均处于禁用状态。这样是为了使用户在代码学习中能深入了解各个功能的具体实现方法。

2．初始化套接字

用户在程序初始化函数中，还应该添加套接字初始化代码，以便程序运行时完成监听端口等功能的准备。代码如下：

```
BOOL CMyDlg::OnInitDialog()
{
    CDialog::OnInitDialog();
    ASSERT((IDM_ABOUTBOX & 0xFFF0) == IDM_ABOUTBOX);
    ASSERT(IDM_ABOUTBOX < 0xF000);
    CMenu* pSysMenu = GetSystemMenu(FALSE);
    ...                                         //省略部分代码
char t[100];
DWORD ss=MAKEWORD(2,0);                         //初始化套接字库
  ::WSAStartup(ss,&data);
SOCKET s;                                       //定义套接字对象
sockaddr_in addr;                               //定义网络地址结构对象
addr.sin_family=AF_INET;                        //为地址结构中的成员赋值
    addr.sin_port=htons(100);
    ::gethostname(&t);                          //获得本机名
   hosent *host=gethostbyname(&t);              //从本机名获取主机信息结构
  addr.sin_addr.S_un.S_addr= inet_addr(host->h_addr_list);
                                                //设置网络字节顺序的 IP 地址
    s=::socket(AF_INET,SOCK_STREAM,IPPROTO_TCP);    //创建套接字
::bind(s,(sockaddr *)addr,sizeof(addr));            //绑定套接字
:: listen(s,3);                                     //监听套接字
:: WSAAsyncSelect(s, this->m_hWnd,WM_SOCKT,FD_READ|FD_ACCEPT);
                                                    //设置异步套接字
...                                                 //省略部分代码
   }
```

在代码中，用户可以看到程序调用函数 WSAAsyncSelect()将监听套接字设置为异步模式。当有相关的套接字事件发生时，程序便会向窗口发送自定义消息 WM_SOCKT。消息 WM_SOCKT 是在工程头文件中定义的自定义消息。代码如下：

```
#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000
#define WM_SOCKT WM_USER+1                          //定义自定义消息
class CMyDlg : public CDialog
{
   ...                                              //省略部分代码
}
```

用户在头文件中，除了定义自定义消息以外，还需要定义消息响应函数。代码如下：

```
class CMy2Dlg : public CDialog
{
public:
    CMy2Dlg(CWnd* pParent = NULL);      // standard constructor
   CString ip;
   SOCKET s;                            //服务器监听套接字
   SOCKET s1;                           //服务器数据收发套接字
int port;
HFILE hfile;
```

```
char buff[100];
        CString str;
        char str12[100];
        HWND h;
    protected:
    virtual void DoDataExchange(CDataExchange* pDX);
public:
    HICON m_hIcon;
sockaddr_in addr;
WSADATA data;
int i;
int n;
   afx_msg void Onsockt1(WPARAM wParam,LPARAM lParam);
                                            //套接字消息响应函数
    afx_msg void OnTimer(UINT nIDEvent);
    afx_msg void OnClose();
...                                         //省略部分代码
}
```

然后，用户还需要在消息映射宏中添加套接字消息的消息映射项。代码如下：

```
...                                         //省略部分代码
BEGIN_MESSAGE_MAP(CMy2Dlg, CDialog)
    //{{AFX_MSG_MAP(CMy2Dlg)
    ON_WM_SYSCOMMAND()
    ON_WM_PAINT()
    ON_WM_QUERYDRAGICON()
    ON_BN_CLICKED(IDC_BUTTON2, OnButton2)
    ON_BN_CLICKED(IDC_BUTTON3, OnButton3)
    ON_BN_CLICKED(IDC_BUTTON1, OnButton1)
    ON_MESSAGE(WM_SOCKT,Onsockt1)           //添加消息映射项
    ON_WM_TIMER()
    ON_WM_CLOSE()
    //}}AFX_MSG_MAP
END_MESSAGE_MAP()
...                                         //省略部分代码
```

用户在工程中添加上面的代码以后，已经为自定义套接字消息添加了消息响应函数。下面，将向用户介绍文件接收和发送功能的实现方法。

8.5.4　代码分析

在服务器端，用户所要实现的功能是接收客户端发送的文件，向客户端发送文件以及与客户端进行文字通信。下面将分别向用户讲解这 3 个功能的实现方法。

1．接收文件

在服务器对话框中，如果程序检测到客户端发送的数据到来时，会将界面中接收按钮设置为可用状态。服务器接收文件的功能代码是在套接字消息响应函数 Onsockt1()中进行实现，处理的套接字事件为 FD_READ。代码如下：

```
void CMyDlg::Onsockt1(WPARAM wParam,LPARAM lParam)
 {
     switch(lParam)

 {
```

```
case FD_READ:                                           //设置读取事件
GetDlgItem(IDC_SAVE)->EnableWindow(true);               //使用保存文件按钮
GetDlgItem(IDC_CLEAR)->EnableWindow(true);              //使用保存文件按钮
     s1=::accept(s,( sockaddr *)addr2,sizeof(addr2);    //应答连接请求
        }
 }
```

当程序在套接字消息响应函数中检测到相应的套接字事件时，将对话框中接收文件按钮设置为可用状态。这时，用户可以通过文件保存对话框 CFileDialog 类，为将要接收的文件选择其保存路径。CFileDialog 类是 MFC 中的一个类，其负责显示文件打开或者保存对话框。在本例中，用户需要使用文件保存对话框，所以，程序中使用该类显示文件保存对话框。代码如下：

```
CFileDialog filedlg(false);     //定义文件对话框对象，并将其指定为文件保存对话框
filedlg.DoModal();              //显示文件保存对话框
```

如果用户将其构造函数的参数设置为 false，则表示用户打开的是一个文件保存对话框。否则，程序将显示文件打开对话框。文件保存对话框如图 8.7 所示。

文件对话框显示以后，用户可以通过其成员函数 GetPathName()获取保存路径。如果该函数调用成功，则返回文件的路径名，返回类型为字符串类型。该函数原型如下：

```
CString GetPathName()const;
```

在工程中，用户可以通过双击控件为“接收文件”按钮添加消息响应函数，并将响应函数名称设置为 OnSave()，如图 8.8 所示。

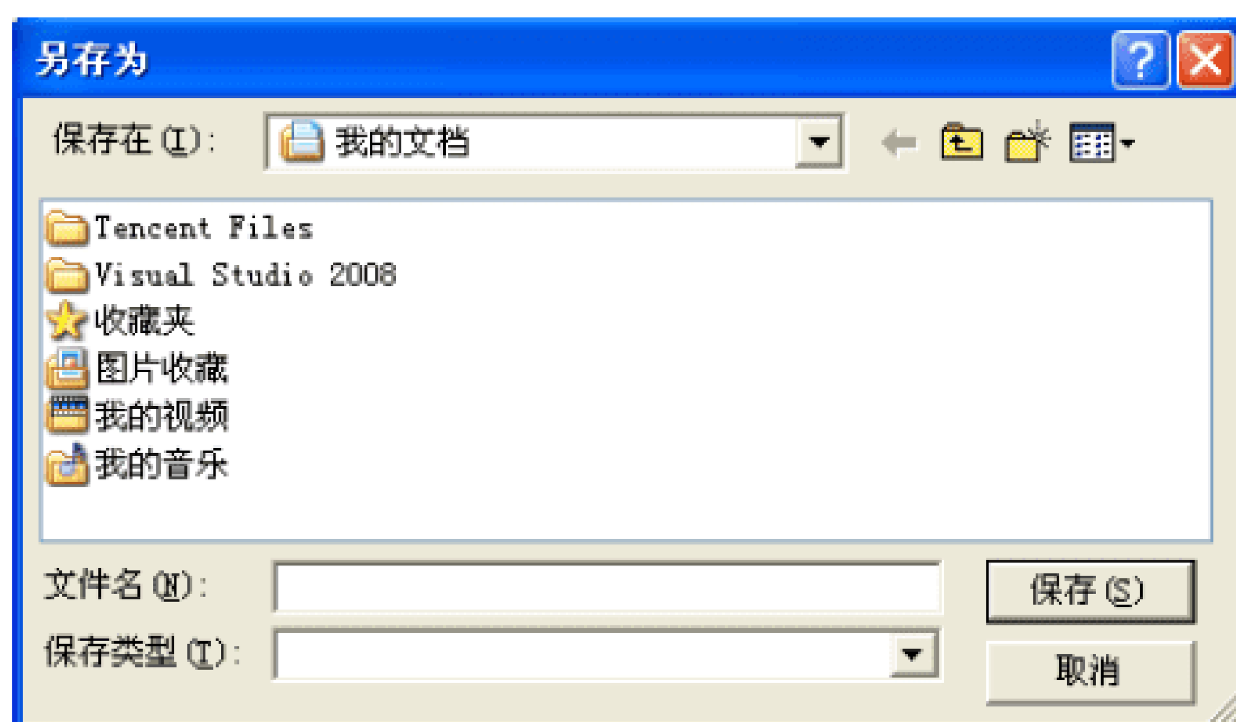


图 8.7　文件保存对话框

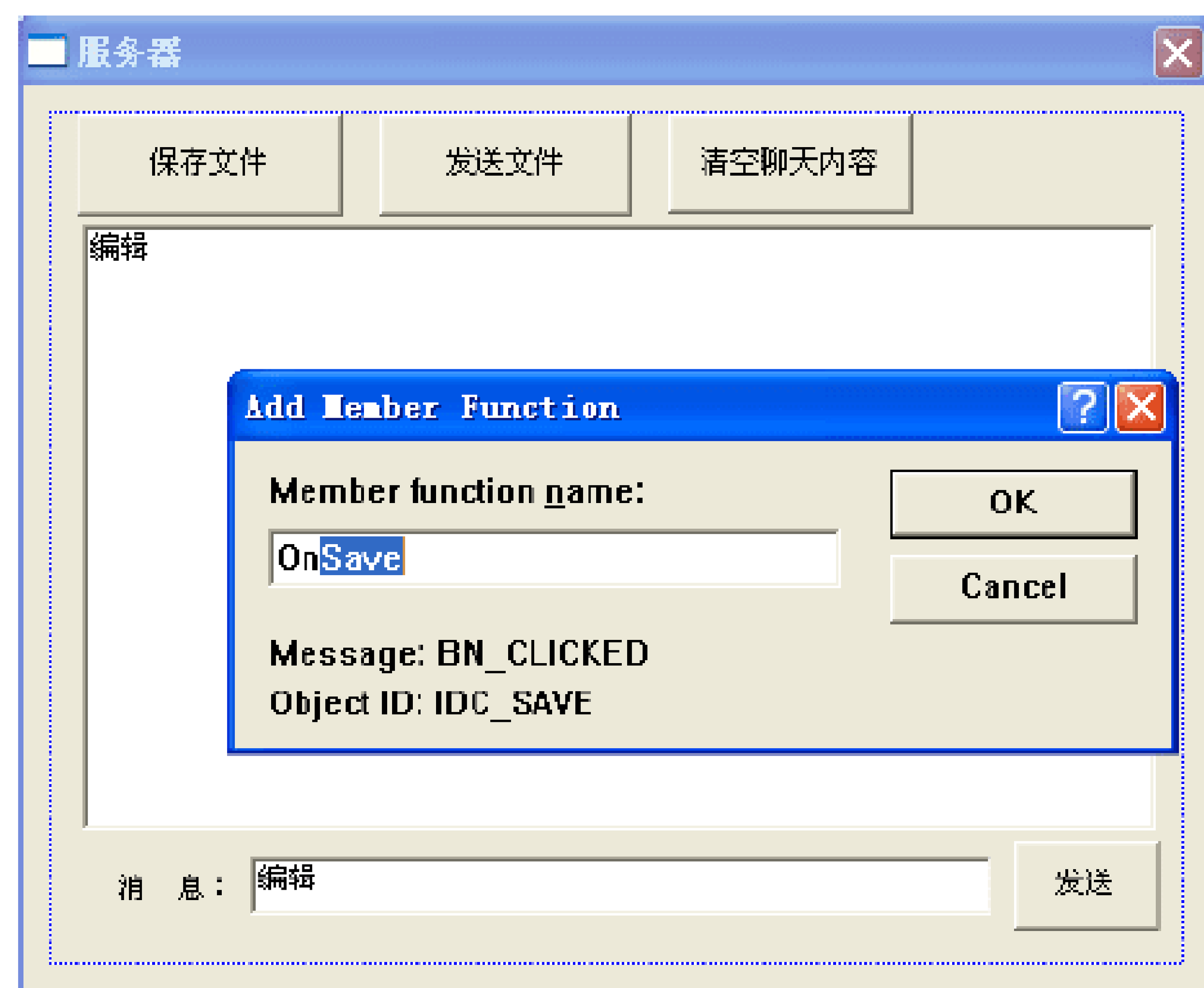


图 8.8　为“接收文件”按钮添加消息响应函数

用户单击 OK 按钮，完成响应函数的添加以后，便可以在函数 OnSave()中，添加服务器接收并保存文件的相关代码。代码如下：

```
void CMyDlg::OnSave()
{
if(s1!=NULL)
{
  ::recv(s1,&text,100);                     //接收文件名
  if(text!=0)
  {
```

```
    CFileDialog filedlg(false);                      //定义文件对话框对象
filedlg.m_ofn.lpstrFileTitle=(LPSTR)text;            //在对话框中显示接收到的文件名
if(filedlg.DoModal()==IDOK)                          //显示文件保存对话框
{
CString str;                                         //定义字符串变量
     str=filedlg.GetPathName();                      //获取文件保存路径
str+=(LPSTR)text;                                    //连接文件路径和文件名
    CFile file(str, CFile::modeReadWrite);           //根据文件名创建文件
    if(file!=NULL)                                   //判断文件是否创建成功
    {
      while(text!=0)                                 //循环接收数据
        {
                ::recv(s1,&text,100);                //接收数据
file.Write(&text,100);                               //写入文件
}
file.Close();                                        //关闭文件或套接字句柄
GetDlgItem(IDC_SAVE)->SetWindowText("发送完成");//提示用户接收数据完成
      }
    }
}
}
}
```

在代码中，用户首先接收文件名，然后根据该文件名创建文件并且将文件路径设置为指定路径。文件创建成功，则向该文件中循环写入数据。最后，关闭文件以及套接字句柄。

2．发送文件

当服务器用户需要向客户端发送文件时，可以通过发送文件按钮执行该功能。首先，使用应用程序向导对话框为其添加消息响应函数 Onliulan()，如图 8.9 所示。

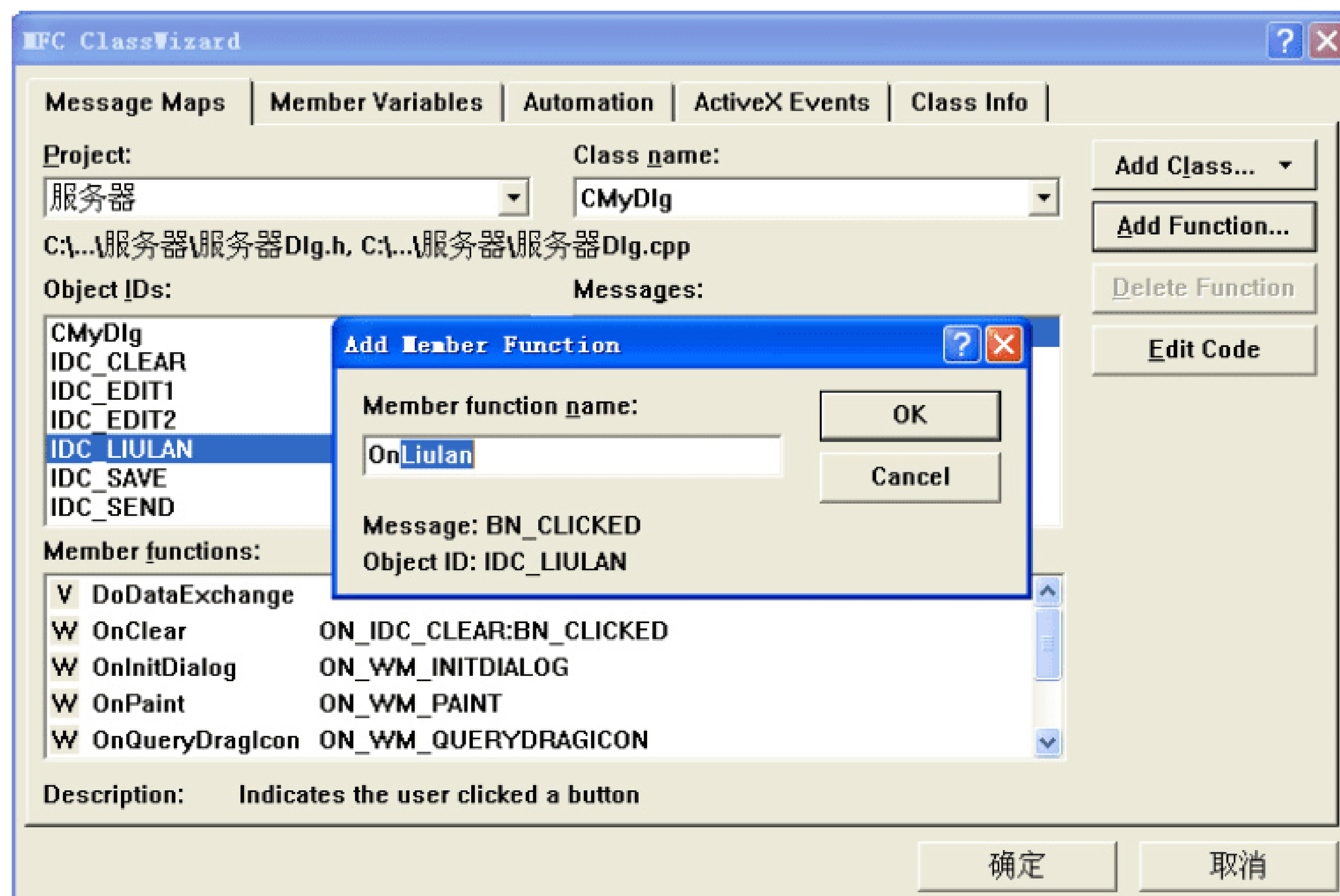


图 8.9　为发送文件按钮添加消息响应函数

然后，在该响应函数中，使用“打开”对话框打开指定文件，并将其发送到相应的客户端，如图 8.10 所示。该功能代码如下：

```
void CMyDlg::OnLiulan()
{
char text[100];                                        //定义字符数组
CFileDialog file(true);                                //定义文件对话框对象
if(file.DoModal()==IDOK)                               //显示文件保存对话框
{
CString str;                                           //定义字符串变量
str=file.GetFileTitle();                               //获取文件名
       ::send(s,str.GetBuffer(1),sizeof(str));         //发送文件名到服务器
      str=file.GetPathName();                          //获取文件保存路径
      CFile file1(str,CFile::modeReadWrite);           //创建文件对象
      file1.Read(text,100)                             //读取文件数据
       while(text!=EOF)                                //判断文件是否结束
     {
file1.Read(text,100);                                  //读取 100 字节的文件数据
       ::send(s1,&text,100);                           //发送文件数据到服务器
          }
GetDlgItem(IDC_LIULAN)->SetWindowText("发送完成");     //提示用户发送数据完成
file1.Close()                                          //关闭文件
}
}
```

用户通过以上代码实现了服务器向客户端发送文件数据的基本功能。用户可以根据实例程序的思路添加或修改代码，以实现自定义的发送文件代码。

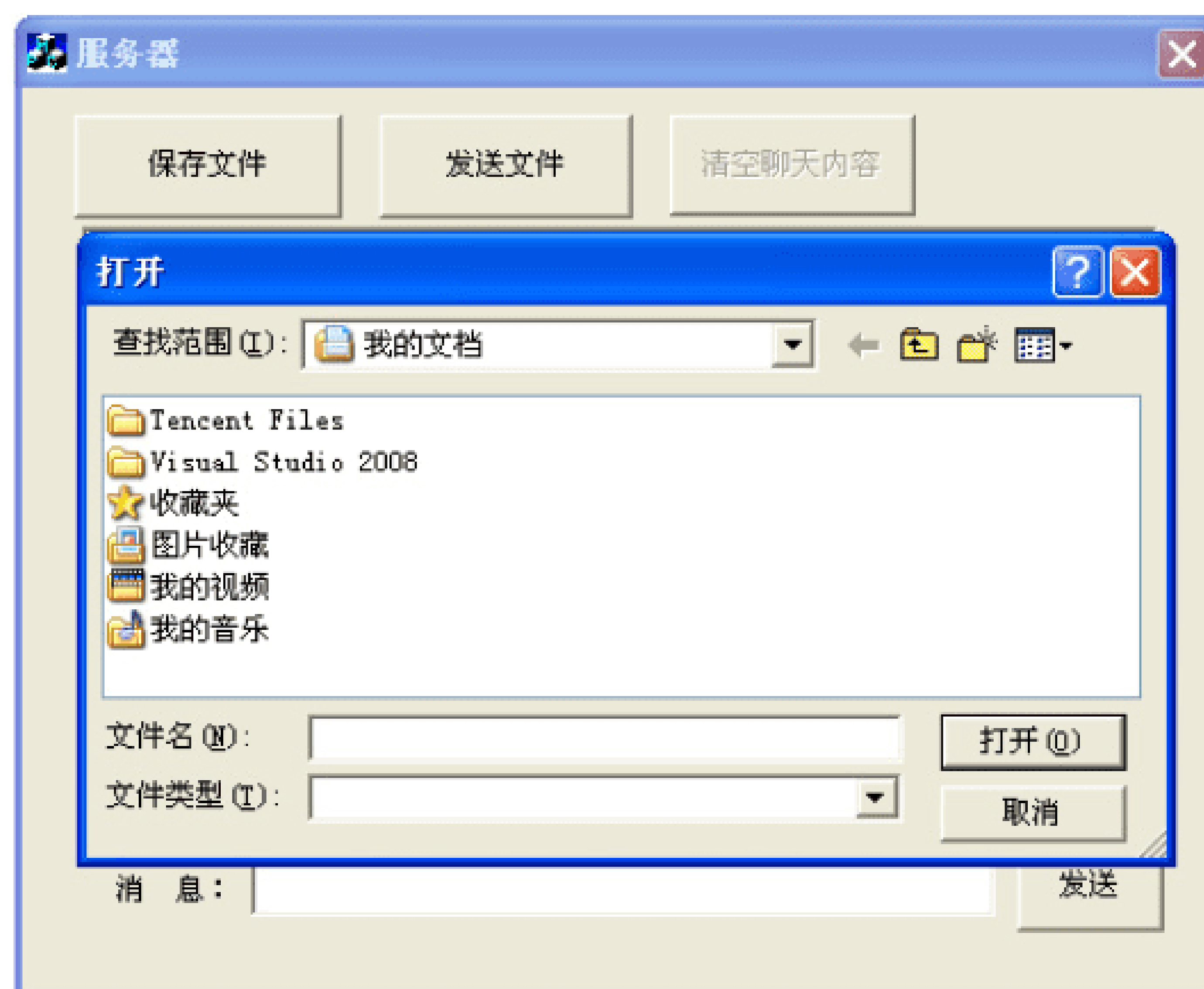


图 8.10 文件打开对话框

3. 文字通信

在实例程序中，服务器与客户端的通信方式都是依靠文件进行传输。因此，服务器端还需要实现文字通信的相关功能代码。其接收代码如下：

```
int n=0;                                               //定义全局变量 n
SOCKADDR_IN add;                                       //套接字地址对象
CString str13;
void CMyDlg::Onsockt1(WPARAM wParam,LPARAM lParam)
 {
char cs[100],cs1[10000],name[15];
```

```
    switch (lParam)
{
      case FD_ACCEPT:                                    //处理连接请求
            {
                s1=::accept(s,NULL,NULL);                //接受客户端的连接请求
         n=n+1;                                          //计数
         str13.Format("有%d 客户已经连接上了",n);         //格式化字符串
       this->SetWindowText(str13);
        GetDlgItem(IDC_EDIT1)->GetWindowText((LPSTR)cs1,10000);
        ::getpeername(s1,(SOCKADDR*)&add,(int*)sizeof(add));
                                                         //获得连接对方的 IP 地址
        str13+=cs1;
        str13+="\r\n";
        str13+=::inet_ntoa(add.sin_addr);                //转换主机字节的 IP
        str13+="登录";
        GetDlgItem(IDC_EDIT1)->SetWindowText(str13);
            }
            break;
        case FD_READ:                                    //处理读取事件
              {
                 GetDlgItem(IDC_SAVE)->EnableWindow(true);
                                                         //使用保存文件按钮
GetDlgItem(IDC_CLEAR)->EnableWindow(true);               //使用保存文件按钮
                s1=::accept(s,( sockaddr *)addr2,sizeof(addr2);
                                                         //应答连接请求
CString num="";
                recv(s1,cs,100,NULL);                    //接收数据
                GetDlgItem(IDC EDIT1)->GetWindowText((LPSTR)cs1,10000);
                //GetDlgItem(IDC EDIT2)->GetWindowText((LPSTR)cs,100);
                      num+=(LPSTR)cs1;
                        num+="\r\n";
                        num+=::inet ntoa(add.sin addr);
                                                         //将 IP 转换为主机顺序
                        num+="对您说: ";
                         num+=cs;
                   GetDlgItem(IDC EDIT1)->SetWindowText(num);}
            break;
}
}
```

注意：由于在本书第 6 章中讲解了关于该功能的具体实现方法，所以在本章中不再进行讲解，请用户复习第 6 章中的相关内容。

服务器除了可以接收客户端的信息以外，还可以发送信息到客户端。代码如下：

```
void CMyDlg::OnSend()
{
char sever[100];                                         //声明字符数组变量
    GetDlgItem(IDC_EDIT2)->GetWindowText((LPSTR)sever,100);
                                                         //获取要发送的数据
    CString str="",str1="";                              //声明字符串变量
    GetDlgItem(IDC_EDIT1)->GetWindowText(str);           //获取发送框中的数据
    if(str!="")                                          //判断发送数据是否为空
    {str+="\r\n";}                                       //添加换行符
    GetDlgItem(IDC_EDIT2)->GetWindowText(str1);
    str+=str1;
```

```
    GetDlgItem(IDC_EDIT1)->SetWindowText(str);      //设置程序界面的显示
    send(s1,sever,100,0);                           //发送数据
}
```

如果服务器与客户端之间通信的内容过多，会导致编辑框中显示的内容也过多。所以，为了方便用户查看消息，应该在程序中实现清除编辑框中所有内容的功能。代码如下：

```
void CMyDlg::OnClear()
{
        GetDlgItem(IDC_EDIT1)->SetWindowText(" ");     //清空数据
}
```

关于服务器的所有功能已经全部实现，如果用户需要为其添加其他功能，可以通过修改本章中的实例代码。用户阅读本章中的代码时，请参考随书光盘中相应章节的代码运行结果，以便跟随笔者的思路进行学习。

8.6 客户端代码

客户端向服务器发送消息或者文件时，首先应该向服务器发送连接请求并等待其应答。然后，再将本地文件读取到指定缓冲区中保存。最后，通过连接套接字发送到服务器。本节将向用户介绍客户端功能代码的编写方法。

8.6.1 客户端功能

本章实例程序中，客户端可以按照指定的服务器地址或端口连接服务器。当客户端连接服务器成功以后，便可以向该服务器发送消息、文件以及接收并保存服务器发送的文件等。其中，发送文件和接收保存文件是客户端的主要功能。因此，在本节中将主要向用户介绍客户端发送文件以及保存文件等功能的具体实现方法。

8.6.2 创建客户端

与服务器端创建方式一样，在 VC 中通过应用程序向导创建客户端界面。该界面同样基于对话框模式。

1. 创建工程

在 VC 编译器中，创建客户端对话框的步骤如下：

（1）选择“文件”|“新建”命令，打开“新建”对话框，并将工程名称修改为“客户端”，如图 8.11 所示。

（2）单击“确定”按钮，进入应用程序类型设置，将该类型设置为对话框模式，如图 8.12 所示。

（3）单击“下一步”按钮，设置该工程应该包含 Windows Sockets 的支持，如图 8.13 所示。

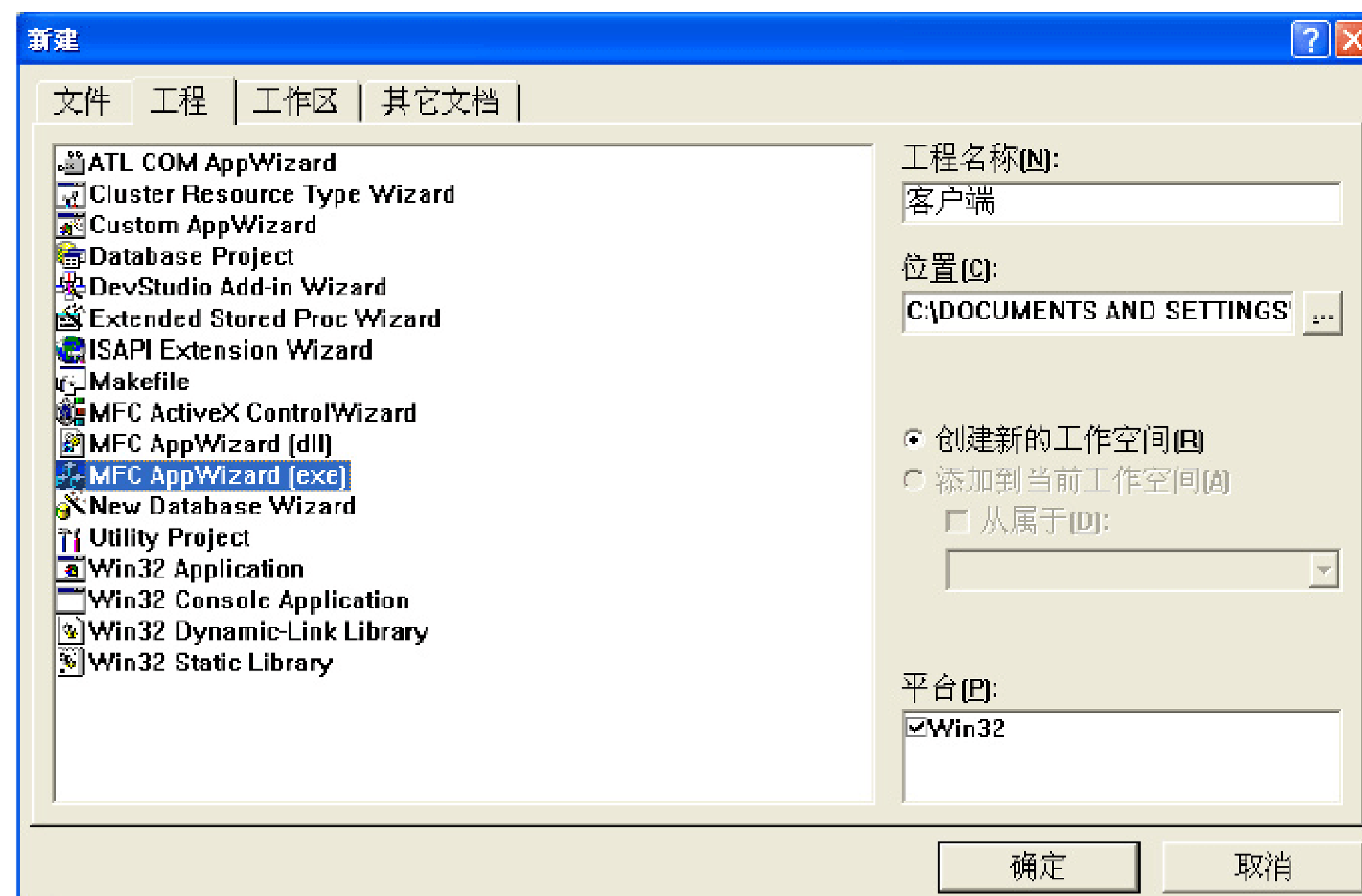


图 8.11　新建客户端工程

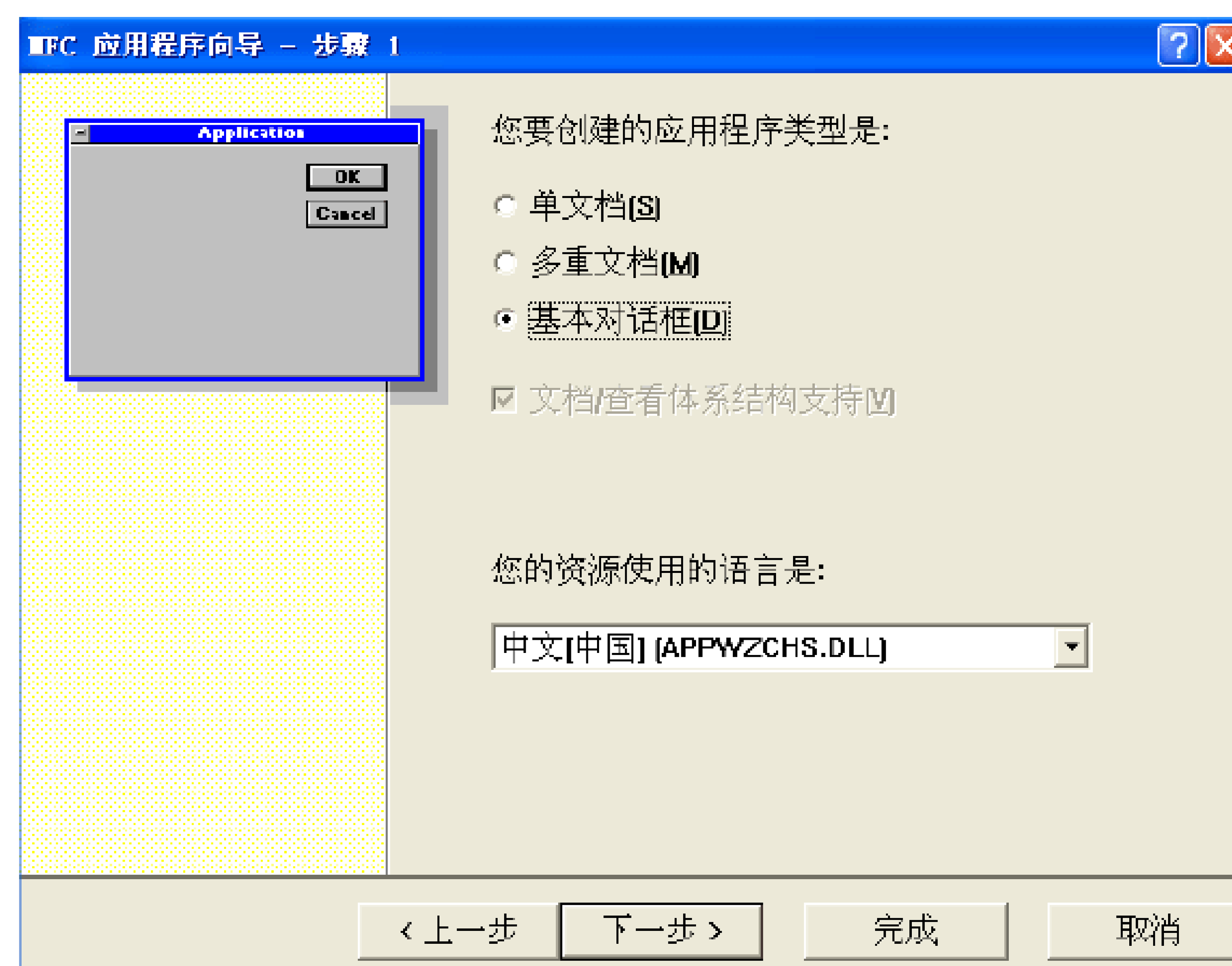


图 8.12　指定应用程序类型



图 8.13　设置工程支持 Windows Sockets

（4）单击“完成”按钮，完成客户端工程的相关设置，并回到 VC 主界面中进行代码编辑。

2．设计界面

用户可以根据客户端的基本功能对其界面进行设计，以适应其功能的需要。根据本章中客户端所具有的相关功能，本节为实例程序所设计的客户端界面如图 8.14 所示。



图 8.14　客户端界面

在客户端界面中，使用的控件 ID、属性以及作用如表 8.11 所示。

表 8.11　客户端控件ID以及属性

控件 ID	控 件 属 性	控 件 作 用
IDC_SAVE	按钮	保存接收到的文件
IDC_LIULAN	按钮	发送文件的位置
IDC_CLEAR	按钮	清空信息
IDC_EDIT1	编辑框	显示服务器和客户端之间聊天信息
IDC_STATIC	静态控件	标识作用
IDC_EDIT2	编辑框	服务器将发送的消息
IDC_SEND	按钮	发送消息

用户从表 8.11 中，可以知道在客户端界面中，相关控件的 ID 以及作用等信息。

8.6.3　界面初始化

在客户端界面中，部分控件应当在程序需要使用时才从禁用状态转为可用状态。这样，从很大程度上避免了用户的操作不当。同时，对于用户的学习也会有很大帮助，让用户明白程序的工作原理。代码如下：

```
BOOL CMyDlg::OnInitDialog()
{
 CDialog::OnInitDialog();
```

```
ASSERT((IDM_ABOUTBOX & 0xFFF0) == IDM_ABOUTBOX);
ASSERT(IDM_ABOUTBOX < 0xF000);
CMenu* pSysMenu = GetSystemMenu(FALSE);
...                                                 //省略部分代码
SetIcon(m_hIcon, TRUE);
SetIcon(m_hIcon, FALSE);
GetDlgItem(IDC_EDIT1)->EnableWindow(false);         //禁用信息显示窗口
GetDlgItem(IDC_SAVE)->EnableWindow(false);          //禁用保存文件按钮
GetDlgItem(IDC_LIULAN)->EnableWindow(false);        //禁用发送文件按钮
GetDlgItem(IDC_CLEAR)->EnableWindow(false);         //禁用清除按钮
...                                                 //省略部分代码
return TRUE;
}
```

用户在客户端程序初始化函数 OnInitDialog()中，将部分功能按钮禁用。控件被禁用后的界面，如图 8.15 所示。

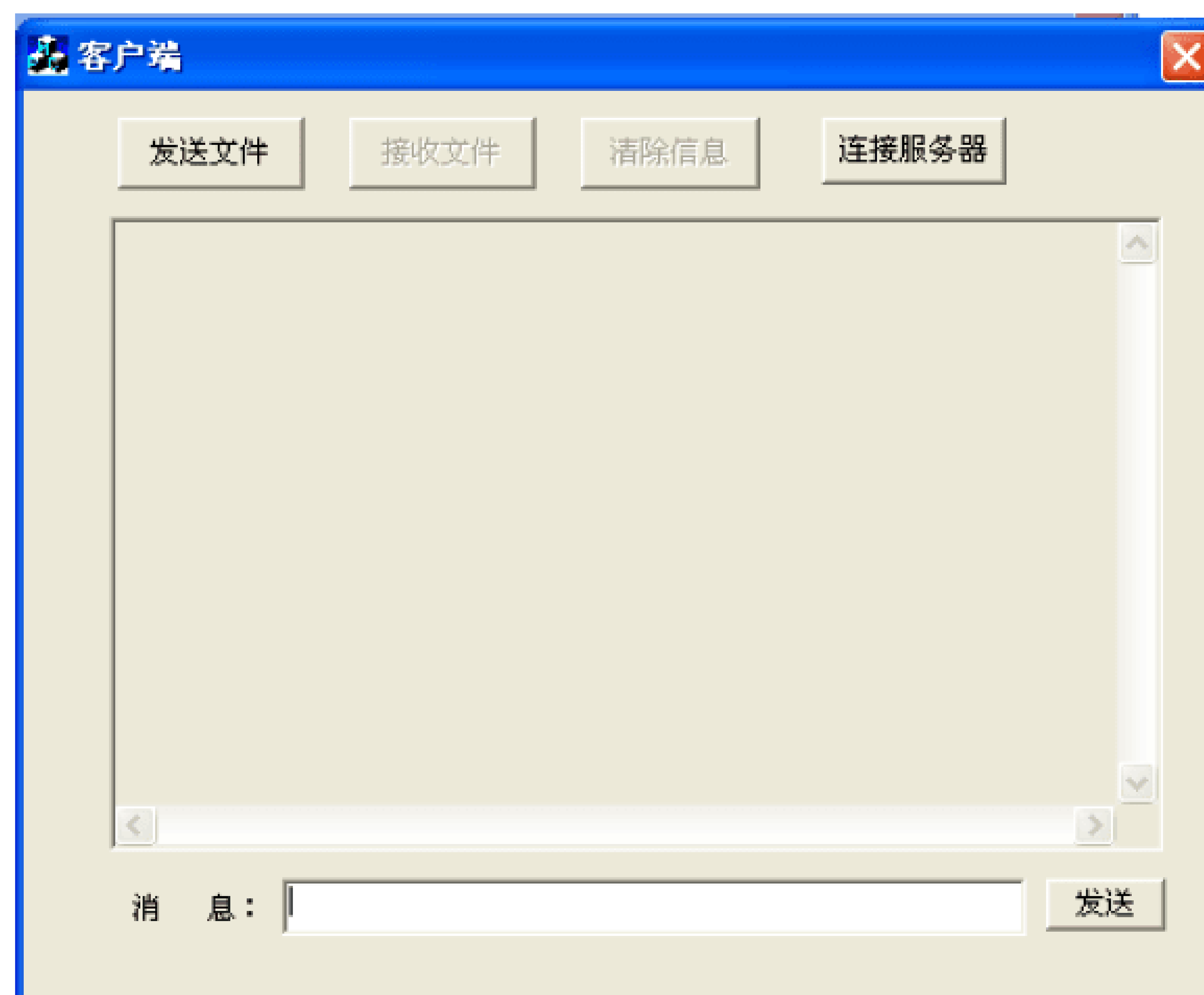


图 8.15　界面初始化

8.6.4　连接服务器

客户端可以通过设置服务器的 IP 地址和端口连接相应的服务器。

1．添加对话框

在客户端初始化时，只有连接服务器按钮处于可用状态。用户单击该按钮会弹出如图 8.16 所示的服务器设置对话框。通过该对话框可以为客户端设置服务器的 IP 地址以及连接端口。

2．使用对话框

如果用户在程序中需要使用到这个对话框，则必须通过应用程序向导为该对话框附加一个类，并指定该类名为 CSSet，如图 8.17 所示。

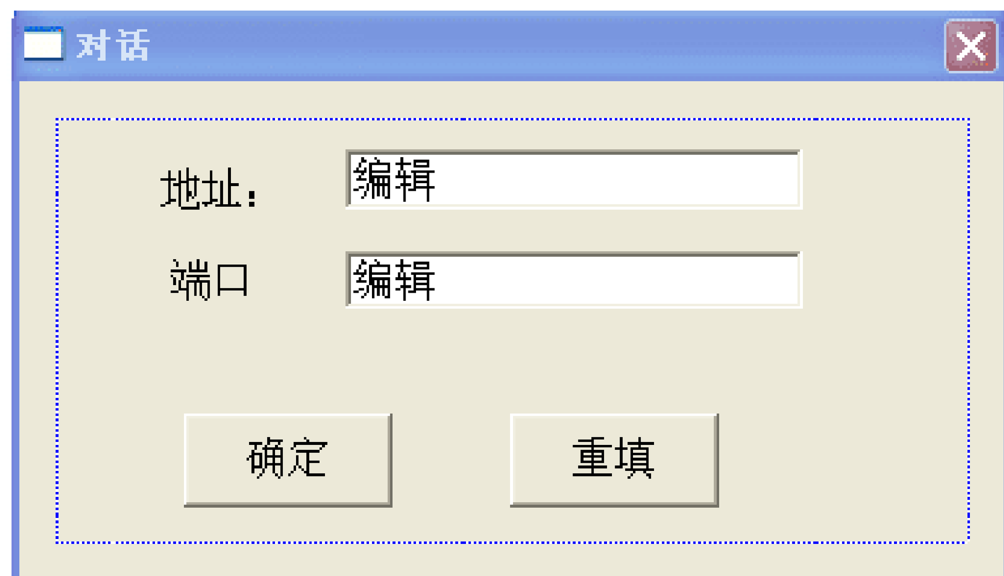


图 8.16　服务器设置对话框

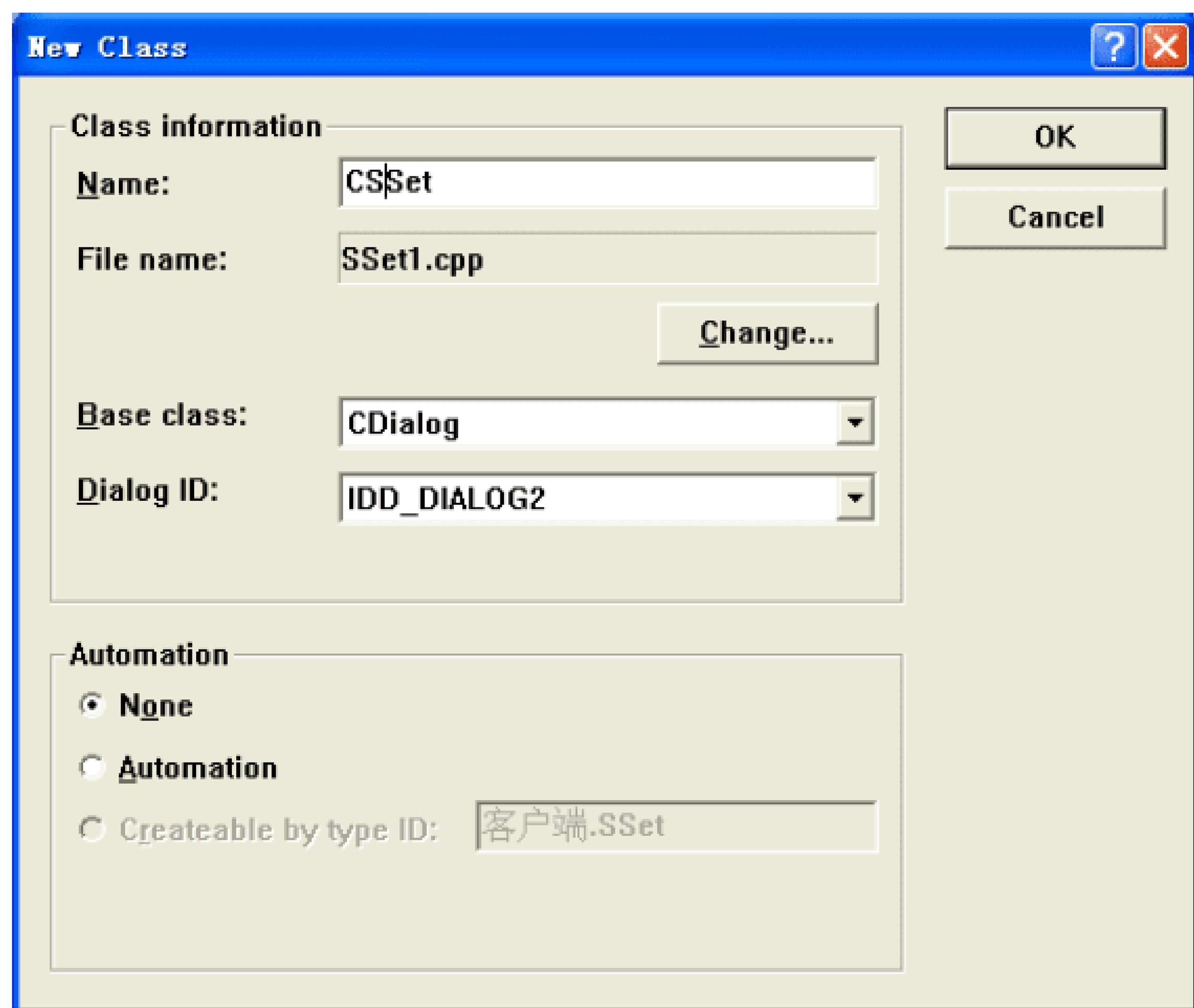


图 8.17　添加新类

打开 CSSet 类的头文件 SSet.h，在该类的声明中定义两个变量，分别表示服务器地址与端口。代码如下：

```
class CSSet : public CDialog
{
public:
    CSSet(CWnd* pParent = NULL);
   CString m_add;              //服务器地址
    int m_port;                //服务器端口号
  ...                          //省略部分代码
  }
```

现在，客户端程序便可以使用对话框类 CSSet。首先，应该在头文件“客户端 Dlg.h”中包含新对话框类的头文件 SSet.h。然后，在主对话框类的声明中定义一个 CSSet 类对象。代码如下：

```
#include "SSet.h"                                  //包含新对话框类的头文件
class CMyDlg : public CDialog
{
public:
    CMyDlg(CWnd* pParent = NULL);
  CSSet set;                                       //定义对象
SOCKET s;                                          //定义套接字对象
sockaddr_in addr;                                  //定义网络地址结构对象
...                                                //省略部分代码
  }
```

在客户端实例界面中，为连接服务器按钮添加消息响应函数 OnConnect()，如图 8.18 所示。

在连接服务器按钮的消息响应函数 OnConnect()中，需要显示 CSSet 类对话框，并且从该对话框中获取服务器的 IP 地址以及端口。代码如下：

```
void CMyDlg::OnConnect()
{
```

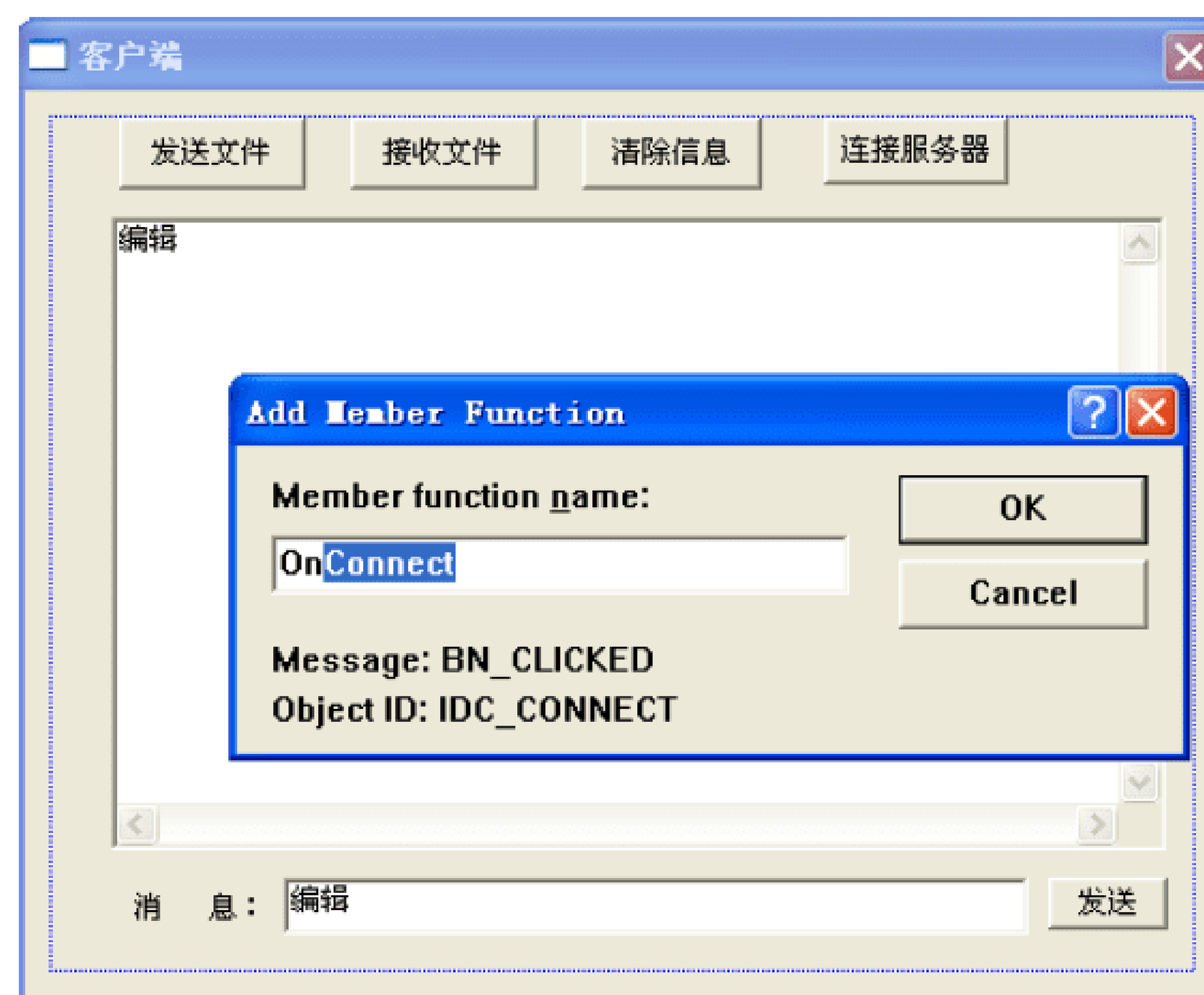


图 8.18　添加响应函数

```
   DWORD ss=MAKEWORD(2,0);                          //初始化套接字库
  ::WSAStartup(ss,&data);
    if(set .DoModal()==IDOK)
    {
      addr.sin_family=AF_INET;                       //为地址结构中的成员赋值
    addr.sin_port=htons(set.m_port);                 //设置服务器端口
    addr.sin_addr.S_un.S_addr=inet_addr(set.m_add);     //转换服务器 IP 地址
    s=::socket(AF_INET,SOCK_STREAM,IPPROTO_TCP);        //创建套接字
:: WSAAsyncSelect(s, this->m_hWnd,WM_CSOCKT,FD_READ);
                                                        //设置异步套接字
if(::connect(s, (sockaddr *)addr, sizeof(addr))!=-1)    //连接服务器
   {                                                 //如果连接服务器成功
        This->SetWindowText("连接成功! ");
GetDlgItem(IDC_LIULAN)->EnableWindow(true);          //界面中主要控件处于可用状态
GetDlgItem(IDC_SAVE)->EnableWindow(true);
GetDlgItem(IDC_CLEAR)->EnableWindow(true);
     }
else                                                 //如果连接服务器失败
{
       This->SetWindowText("连接失败，请重试! ");
GetDlgItem(IDC_LIULAN)->EnableWindow(false);         //禁用程序主要按钮控件
GetDlgItem(IDC_SAVE)->EnableWindow(false);
GetDlgItem(IDC_CLEAR)->EnableWindow(false);
            }
        }
}
```

在代码中，程序调用 API 函数 WSAAsyncSelect()将客户端的连接套接字设置为异步模式。当有相关的套接字事件发生时，程序便会向窗口发送自定义消息 WM_CSOCKT。消息 WM_CSOCKT 是在工程头文件中定义的自定义消息。代码如下：

```
#if  MSC VER > 1000
#pragma once
#endif //  MSC VER > 1000
#define WM_CSOCKT WM_USER+11               //定义自定义消息
```

```
class CMyDlg : public CDialog
{
  ...                                    //省略部分代码
}
```

用户在头文件中，除了定义自定义消息以外，还需要定义消息响应函数。代码如下：

```
class CMy2Dlg : public CDialog
{
public:
...                                              //省略部分自动生成的代码
  CString ip;
  SOCKET s;                                      //服务器监听套接字
int port;                                        //服务器端口号
CFile file;                                      //文件对象
char buff[100];                                  //自定义缓冲区
CString str;                                     //字符串变量
HWND h;                                          //窗口实例句柄
public:
SOCKET s;                                        //连接套接字句柄
sockaddr_in addr;                                //套接字地址信息
WSADATA data;                                    //套接字版本
int i;                                           //循环变量
int n;
 afx_msg void Oncsockt1(WPARAM wParam,LPARAM lParam);
                                                 //套接字消息响应函数
    afx_msg void OnTimer(UINT nIDEvent);
    afx_msg void OnClose();
...                                              //省略部分代码
}
```

然后，用户还需要在消息映射宏中添加套接字消息的消息映射项，以便将套接字消息与其消息响应函数相关联。代码如下：

```
...                                       //省略部分代码
BEGIN_MESSAGE_MAP(CMyDlg, CDialog)        //消息映射开始
//{{AFX_MSG_MAP(CMyDlg)
ON_WM_SYSCOMMAND()
ON_WM_PAINT()
ON_WM_QUERYDRAGICON()
ON_MESSAGEMAP(WM_CSOCKT, Oncsockt1)       //套接字消息映射项
ON_BN_CLICKED(IDC_CONNECT, OnConnect)     //连接按钮的消息映射项
ON_BN_CLICKED(IDC_CLEAR, OnClear)         //清除按钮的消息映射项
ON_BN_CLICKED(IDC_LIULAN, OnLiulan)       //发送文件按钮的消息映射项
ON_BN_CLICKED(IDC_SAVE, OnSave)           //接收文件按钮的消息映射项
ON_BN_CLICKED(IDC_SEND, OnSend)           //发送消息按钮的消息映射项
//}}AFX_MSG_MAP
END_MESSAGE_MAP()                         //结束消息映射
...                                       //省略部分代码
```

用户在文件“客户端 Dlg.cpp”中添加上述代码以后，便为自定义套接字消息 WM_CSOCKT 添加了相应的消息响应函数。下面，将向用户介绍客户端收发消息以及接收发送文件的功能实现，其中，接收文件功能就是在套接字消息响应函数中实现。

8.6.5　代码分析

在客户端中，其代码模块主要分为收发消息、接收文件以及发送文件等。本节将根据顺序向用户分别讲解各个模块的代码实现。

1．收发消息

为了在客户端中实现与服务器进行文字交流，用户必须为客户端实现收发文字消息的功能。首先，为发送按钮添加相应的消息响应函数 OnSend()。然后，在该函数中添加代码以实现消息的收发功能。代码如下：

```
void CMyDlg::OnSend()
{
 char sever[100];                                       //声明字符数组变量
 GetDlgItem(IDC_EDIT2)->GetWindowText((LPTSTR)sever,100);
                                                        //获取要发送的数据
 CString str="",str1="";                                //声明字符串变量
 GetDlgItem(IDC_EDIT1)->GetWindowText(str);             //获取发送框中的数据
 if(str!="")                                            //判断发送数据是否为空
 {
str+="\r\n";
}                                                       //添加换行符
 GetDlgItem(IDC_EDIT2)->GetWindowText(str1);
 str+=str1;
    GetDlgItem(IDC_EDIT1)->SetWindowText(str);          //设置程序界面的显示
    send(s1,sever,100,0);                               //发送数据
}
```

在 VC 中编译运行上面的程序，当客户端连接服务器成功以后，用户便可以发送文字消息到服务器器且将这些文件显示在客户端界面上，如图 8.19 所示。

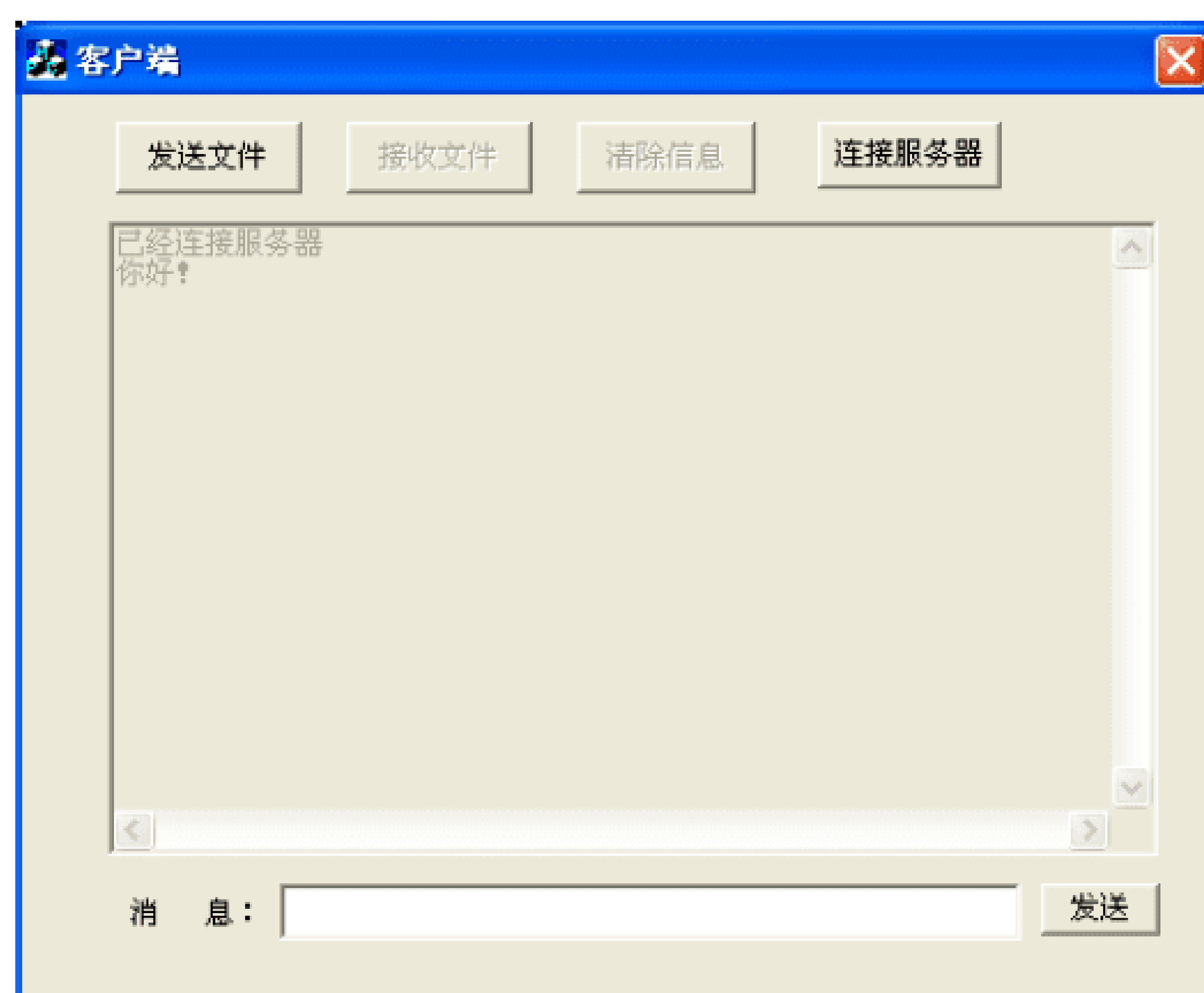


图 8.19　发送消息到服务器

2．接收文件

客户端与服务器一样，具有接收文件的功能。当客户端检测到服务器发送的文件时，

将接收文件按钮设置为可用状态，这时用户可以通过单击该按钮选择接收文件的保存路径。当保存路径选择后，客户端开始接收文件数据。代码如下：

```
void CMyDlg::Oncsockt1(WPARAM wParam,LPARAM lParam)
 {
     switch(lParam)

 {
case FD_READ:                                         //设置读取事件
...                                                   //省略部分代码
GetDlgItem(IDC_SAVE)->EnableWindow(true);             //使用保存文件按钮
GetDlgItem(IDC_CLEAR)->EnableWindow(true);            //使用保存文件按钮
    ...                                               //省略部分代码
        }
 }
```

然后，用户需要为接收文件按钮添加消息响应函数 OnSave()，如图 8.20 所示。

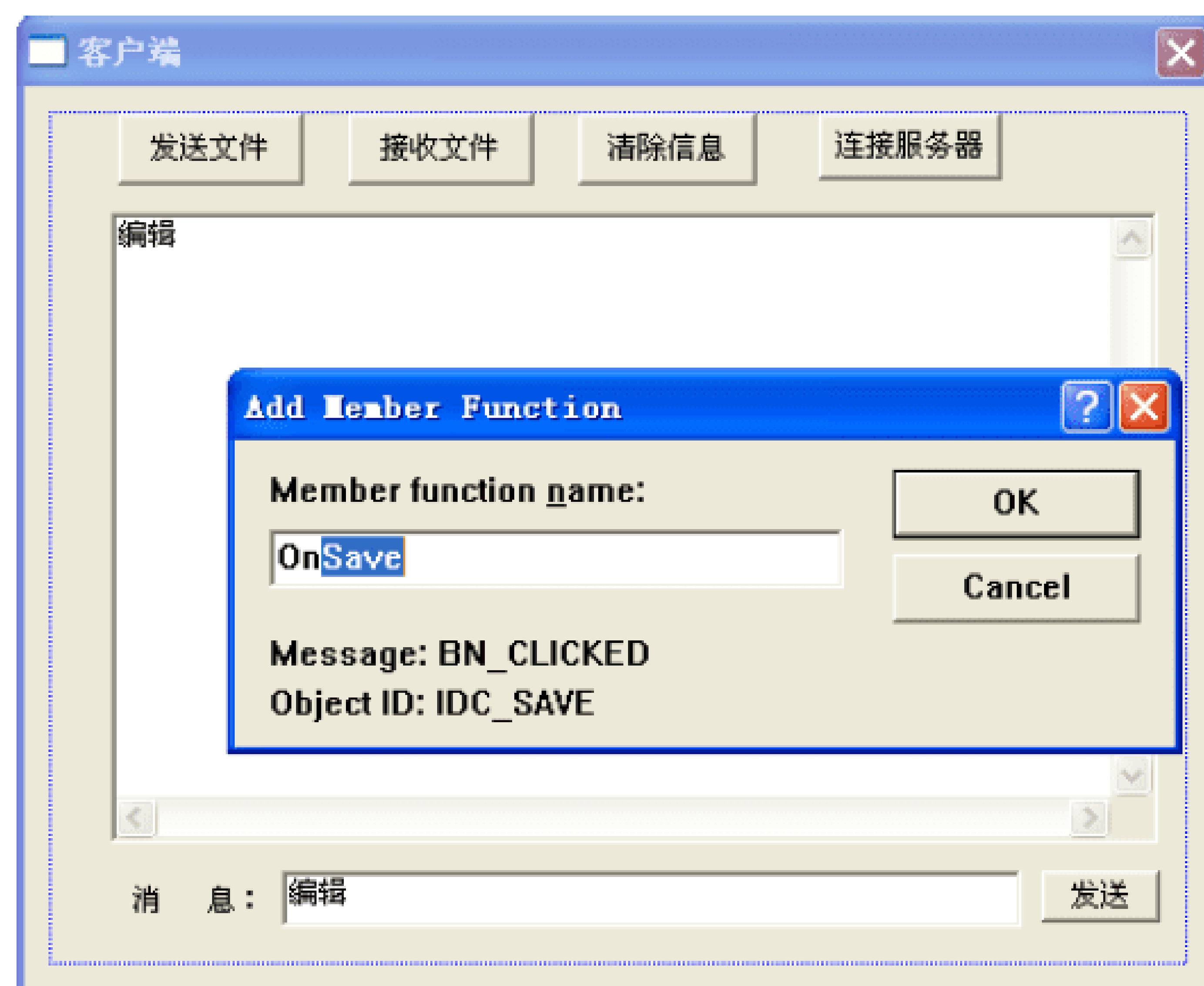


图 8.20　添加消息响应函数

在消息响应函数 OnSave()中，使用文件对话框 CFileDialog 类显示文件保存对话框，实现真正意义上的文件接收功能。代码如下：

```
void CMyDlg::OnSave()                                 //接收文件按钮的消息响应函数
{
if(s!=NULL)
{
  ::recv(s,&text,100);                                //接收文件名
  if(text!=0)
  {
    CFileDialog filedlg(false);                       //定义文件对话框对象
filedlg.m_ofn.lpstrFileTitle=(LPSTR)text;             //在对话框中显示接收到的文件名
if(filedlg.DoModal()==IDOK)                           //显示文件保存对话框
{
CString str;                                          //定义字符串变量
    str=filedlg.GetPathName();                        //获取文件保存路径
str+=(LPSTR)text;                                     //连接文件路径和文件名
    CFile file(str, CFile::modeReadWrite);            //根据文件名创建文件
    if(file!=NULL)                                    //判断文件是否创建成功
```

```
    {
      while(text!=0)                                      //循环接收数据
        {
                ::recv(s,&text,100);                      //接收数据
file.Write(&text,100);                                    //写入文件
}
file.Close();                                             //关闭文件或套接字句柄
GetDlgItem(IDC_SAVE)->SetWindowText("接收完成");          //提示用户接收数据完成
     }
   }
}
}
}
```

用户通过文件保存对话框选择接收文件的保存路径，然后在该路径上创建与接收文件名称相同的空文件。空文件创建成功以后，客户端便开始接收文件数据，并将这些数据保存到刚创建的文件中，直到文件数据接收完毕。用户可以参考随书光盘中的实例代码。

3．发送文件

客户端向服务器发送文件是通过发送文件按钮实现的。当用户单击该按钮后，程序应该弹出“打开”对话框供用户选择将要发送的文件以及其路径。然后，程序根据文件路径创建文件对象并打开该文件进行读取，直到读取全部数据成功。最后，将缓冲区中的数据发送到服务器即可。代码如下：

```
void CMyDlg::OnLiulan()                               //发送文件按钮的消息响应函数
{
char text[100];                                       //定义字符数组
CFileDialog file(true);                               //定义文件对话框对象
if(file.DoModal()==IDOK)                              //显示文件保存对话框
{
CString str;                                          //定义字符串变量
str=file.GetFileTitle();                              //获取文件名
       ::send(s,str.GetBuffer(1),sizeof(str));        //发送文件名到服务器
      str=file.GetPathName();                         //获取文件保存路径
      CFile file1(str,CFile::modeReadWrite);          //创建文件对象
      file1.Read(text,100)                            //读取文件数据
       while(text!=EOF)                               //判断文件是否结束
     {
file1.Read(text,100);                                 //读取 100 字节的文件数据
       ::send(s,&text,100);                           //发送文件数据到服务器
         }
GetDlgItem(IDC_LIULAN)->SetWindowText("发送完成");    //提示用户发送数据完成
file1.Close()                                         //关闭文件
}
}
```

当用户创建 CFileDialog 类对象时，如果将构造函数的参数设置为 true，则程序将显示文件打开对话框，如图 8.21 所示。用户在该对话框中选择将发送的文件，然后程序创建与该文件相关联的文件对象。程序利用该文件对象读取文件数据到缓冲区中保存。最后，将缓冲区中的内容通过套接字发送到服务器完成客户端的发送功能。

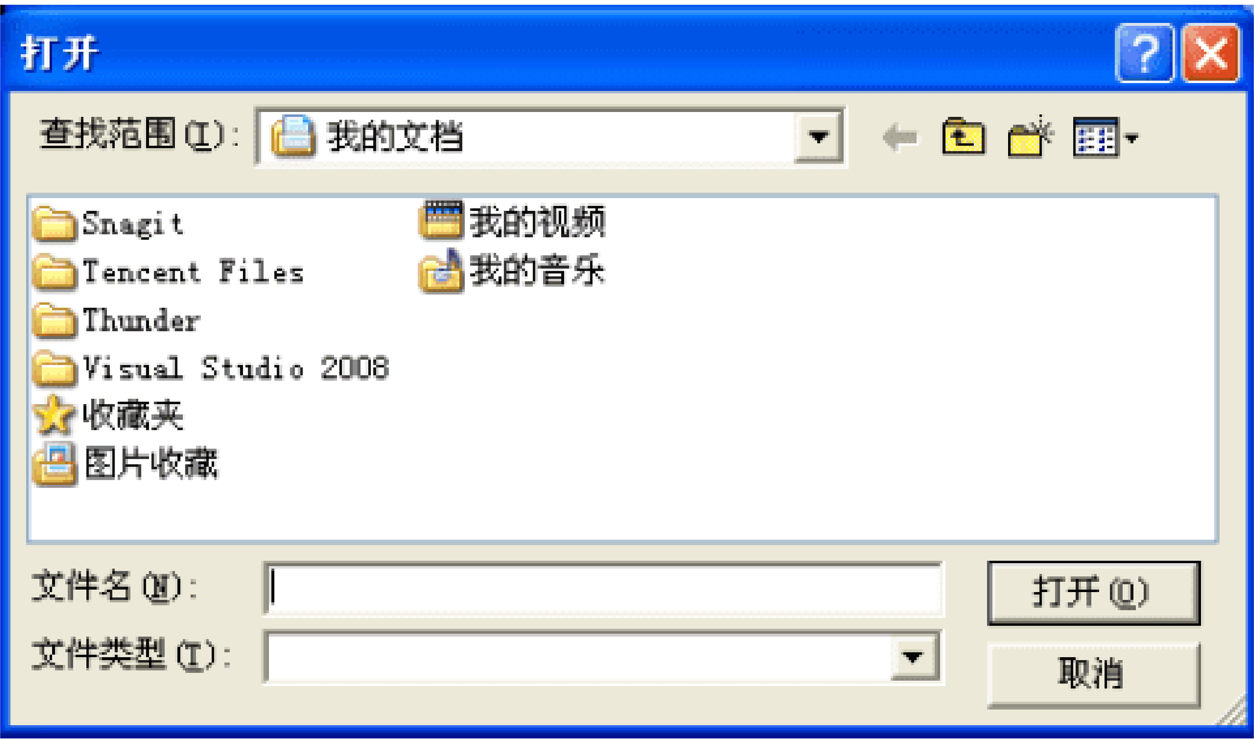


图 8.21　“打开”对话框

8.7　小　　结

在本章中，主要向用户介绍了网络文件传输器的基本原理，结合实例程序分别介绍了传输器的服务器与客户端。在服务器和客户端的功能实现中，主要讲述了服务器的基本功能，并在 VC 开发环境下，通过编写其功能代码向用户讲解服务器端的功能实现。

通过本章对网络文件传输器相关知识的学习，用户应该能基本掌握网络文件传输器的工作原理，并且根据该原理实现服务器与客户端之间的网络连接以及文件收发等。其中，网络连接与文件收发部分均是通过 Windows 套接字实现的，而文件保存以及打开操作是依靠文件对话框类 CFileDialog 实现的。所以，用户应该将随书光盘中的实例代码与程序运行结果相比较学习，这样会使用户的理解更深刻。

第 9 章　实用播放器

在实际生活中，用户所使用的播放器会有很多种，包括专门播放音乐和视频等数据格式的播放器。在本章中，将向用户举例介绍 MP3 的文件格式等基础知识，并且在 VC 中编写一种能够播放 MP3 音乐的播放器。

9.1　播放器编程基础

由于各种音频、视频信息都是以不同的数据格式进行存储的。因此，用户对这些固定数据格式的音频、视频文件进行解码时，需要使用播放器。例如，用户播放 MP3 格式的音乐文件时，播放器将数字信号转换成音频信号后送入音频设备中进行播放。在这个过程中，播放器起着解码的作用。一般情况下，播放器可以分为音乐播放器和视频播放器等种类。在本章中，将以 MP3 音乐文件为例，主要向用户讲解音乐播放器程序的实现方法。

9.1.1　MP3 介绍

MP3 格式是一种音频压缩格式。使用这种格式进行压缩的数据容量较小。一般情况下，MP3 格式是按照 1∶12 的倍率对数据进行压缩。因此，MP3 格式的数据采用的是有损压缩，将大容量的音频数据丢弃部分数据后重新进行组合。在这种情况下，音乐数据的少量丢失并不会影响 MP3 音乐的播放质量。

9.1.2　播放 MP3 文件

在本章中，主要向用户讲解在 VC 中编程实现播放 MP3 音乐的功能。首先，用户需要使用 API 函数或者 MFC 类读取该格式文件，获取相应的文件信息。然后，用户可以使用相关的函数播放 MP3 音乐。

1. 读取MP3音乐文件

与一般文件一样，MP3 音乐文件拥有固定的数据结构。所以，当用户读取 MP3 音乐文件时，可以将该文件当作一般的文件进行读取。MP3 数据结构是由两部分构成：数据帧和标签帧。其中，数据帧包含了该音乐的实体数据。而标签帧中是以字符 TAG 为标记，长度为 MP3 文件的最后 128 字节。在标签帧中包含了该音乐的演唱者、音乐名称以及时间等信息。用户知道这些知识以后，便可以在编程时自定义一个数据结构获取 MP3 音乐文件的相关信息。

🔔注意：由于本章实例中主要是使用 API 函数播放 MP3 音乐，所以关于数据帧的讲解将省略。主要向用户讲解 MP3 文件的标签帧。

例如，在本实例中，用户可以自定义结构体获取 MP3 文件信息。结构定义如下：

```
typdef struct mp3_struct          //自定义 MP3 结构体
{
char heade[3];                    //TAG 字符标记
char title[30];                   //音乐文件名称
char arti [30];                   //演唱者
char alb [30];                    //专辑
char year[4];                     //出版年份
char text[28];                    //备注内容
...                               //省略部分成员
} mp3struct;
```

在该自定义结构中，列出了一些关于 MP3 音乐文件的信息结构成员并省略了 3 个不常用的字节。用户可以根据该结构体编程读取文件相关信息。首先，用户使用 MFC 文件类 CFile 读取文件，再将文件指针移动到该文件最后。然后从文件最后向前读取 128 个字节即可获取到 MP3 文件的相关信息。代码如下：

```
...                                                 //省略部分代码
mp3struct mp3={0};                                  //定义并初始化字符数组
CFile file("C:\\卡门 .mp3",CFile::modeReadWrite|CFile::typeBinary);
                                                    //创建文件对象
file.Seek(-128,CFile::end);                         //从文件结尾处移动文件
                                                    //指针
file.Read(&mp3,128);                                //从文件中读取 128 个字节
MessageBox(mp3.arti);                               //显示歌曲的演唱者
...                                                 //省略部分代码
```

在代码中，用户首先创建与 MP3 文件相关联的文件对象 file，并且指定以二进制方式打开该文件。将上面的代码编译运行之后，程序会弹出对话框并在该对话框上显示该歌曲的演唱者，如图 9.1 所示。

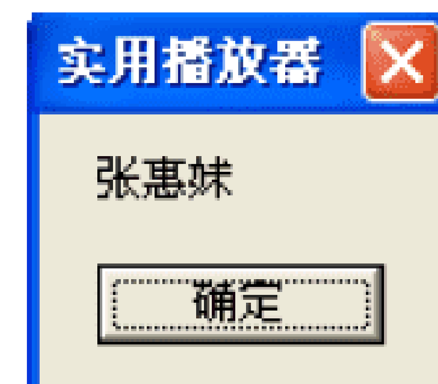


图 9.1　显示歌曲的演唱者

🔔注意：在 CFile 类的函数 Seek()中是将第一个参数设置为于–128，表示文件指针是从文件结尾处向前移动 128。如果用户将该参数设置为正，则函数将返回错误。

如果用户需要显示其他 MP3 文件信息，则仅需将结构体 mp3struct 中的各个成员变量值显示出来即可。例如，显示 MP3 文件的所有相关信息。代码如下：

```
...                                   //省略部分代码
CString str=" ";                      //定义并初始化字符串
str+=mp3.title;                       //显示音乐标题
str+="\r\n";                          //添加换行符
str+=mp3.arti;                        //显示音乐演唱者
str+="\r\n";
str+=mp3.alb;                         //显示音乐的专辑信息
str+="\r\n";
```

```
str+=mp3.year;                                    //显示音乐的出版年份
str+="\r\n";
MessageBox(str);                                  //显示音乐文件相关信息
...                                               //省略部分代码
```

用户将上面的代码进行编译、运行之后，程序弹出对话框并将显示该 MP3 文件的所有信息，如图 9.2 所示。

🔔注意：用户在实际编程中，可以使用本节所定义的 MP3 结构获取文件的相关信息。

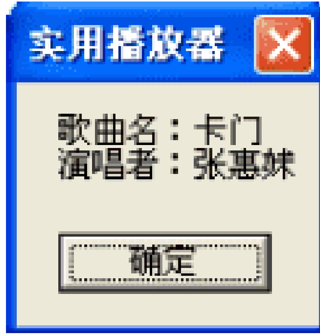


图 9.2　显示 MP3 文件的所有信息

2. 调用函数播放MP3音乐

在上述内容中，用户学习了 MP3 的数据结构知识，通过该数据结构可以获取 MP3 的附加信息。在本小节中，将向用户介绍在 Windows 系统中调用 API 函数播放 MP3 歌曲的函数使用方法。

用户在 VC 中播放 MP3 音乐可以调用 API 函数 mciSendCommand()和 mciSendString()实现。由于在本章中将主要使用函数 mciSendCommand()进行实例编程，所以函数 mciSendString()将不在本章的讲解范围内，请用户参考 MSDN。函数 mciSendCommand()的原型如下：

```
MCIERROR mciSendCommand(
  MCIDEVICEID IDDevice,
  UINT uMsg,
  DWORD fdwCommand,
  DWORD dwParam
);
```

该函数的作用是发送指定功能的命令消息到指定设备执行。其功能十分强大，不仅可以实现播放 MP3 的功能，还可以播放其他格式的音视频数据或操作一些硬件设备。例如，光驱等。该函数如果调用成功，则返回 0。否则，函数将返回相关错误信息。函数具有 4 个参数，意义分别如下：

- ❑ 参数 IDDevice 表示 MCI 设备 ID。

🔔注意：用户在首次打开该设备时，使用系统默认的多媒体设备，则可以直接设置为 0。否则，MCI 可用的设备如表 9.1 所示。

表 9.1　MCI可用设备取值

MCI可用设备取值	意　　义
MCI_DEVTYPE_ANIMATION	动画播放设备
MCI_DEVTYPE_CD_AUDIO	CD音频播放设备
MCI_DEVTYPE_DAT	数字音频设备
MCI_DEVTYPE_WAVEFORM_AUDIO	波形音频设备
MCI_ALL_DEVICE_ID	包括系统中所有的MCI设备
MCI_DEVTYPE_DIGITAL_VIDEO	数字视频设备
MCI_DEVTYPE_OTHER	未定义的设备

当用户播放 MP3 等常用格式的音频文件时，需要将该参数值设置为 0 或者是 MCI_DEVTYPE_WAVEFORM_AUDIO 即可。

- 参数 uMsg 表示发送到设备的命令消息。其取值如表 9.2 所示。

表 9.2　设备命令消息

命令消息	意　义	命令消息	意　义
MCI_OPEN	打开命令	MCI_CLOSE	关闭命令
MCI_PLAY	播放命令	MCI_PAUSE	暂停命令
MCI_STOP	停止命令	MCI_RESUME	恢复命令

注意：在表 9.2 中列出的设备命令消息都是用户常用的一些设备命令消息，如果用户需要了解更多关于 MCI 的设备命令消息请参考 MSDN，在本书中不再对其进行赘述。

- 参数 fdwCommand 表示命令消息的标志位。其取值如表 9.3 所示。

表 9.3　命令消息标志位取值

取　值	意　义
MCI_OPEN_ELEMENT	当用户执行MCI打开命令MCI_OPEN时，必须指定该标志位。如果用户指定该标志位，那么命令参数结构必须为MCI_OPEN_PARMS，用户可以在此结构中指定播放文件的设备类型以及文件名等
MCI_FROM	当用户执行命令MCI_PLAY时，必须指定该标志位，表示从文件起始位置播放文件。其命令参数结构为MCI_PLAY_PARMS
MCI_SEEK_TO_START	当用户执行命令MCI_START时，必须指定该标志位
MCI_TO	当用户执行命令MCI_SEEK时，必须指定该标志位。其命令参数结构为MCI_SEEK_PARMS
MCI_SET_DOOR_OPEN	表示打开光驱。当用户执行命令MCI_SET时，必须指定该命令消息标志位，其命令参数结构为NULL
MCI_SET_DOOR_CLOSED	表示关闭光驱。当用户执行命令MCI_SET时，关闭光驱操作所要指定的命令消息标志位。其命令参数结构为NULL

注意：在表 9.3 中列举了部分常用 MCI 命令消息标志位。如果用户需要了解其他命令消息的标志位可以参考 MSDN，本章不再赘述。

- 参数 dwParam 表示 MCI 命令参数结构的地址。一般情况下，用户可以在该参数所指向的结构中设置 MCI 操作的相关信息。例如，在表 9.3 中所提到的命令参数结构体 MCI_OPEN_PARMS 和 MCI_PLAY_PARMS 是用户在实际编程中会经常使用到的命令参数结构体。常用结构体定义如下：

```
typedef struct {                                    //结构体 MCI_OPEN_PARMS 定义
    DWORD  dwCallback;                              //回调函数地址
    MCIDEVICEID  wDeviceID;                         //设备 ID
    LPCSTR  lpstrDeviceType;                        //设备类型
    LPCSTR  lpstrElementName;                       //文件名
    LPCSTR  lpstrAlias;
} MCI_OPEN_PARMS;
```

```
typedef struct {                          //结构体 MCI_PLAY_PARMS 定义
    DWORD dwCallback;                     //回调函数地址
    DWORD dwFrom;                         //播放开始位置
    DWORD dwTo;                           //播放结束位置
} MCI_PLAY_PARMS;
```

例如，用户使用结构体 MCI_OPEN_PARMS 打开或关闭光驱。代码如下：

```
...                                                //省略部分代码
MCIDEVICEID mci;                                   //MCI 设备 ID 对象
MCI_OPEN_PARMS open;                               //结构体变量
open.lpstrDeviceType = "CDAUDIO";                  //指定设备类型为 CD-ROM
mciSendCommand(NULL,MCI_OPEN,MCI_WAIT|MCI_OPEN_TYPE,(DWORD)// open);
                                                   //初始化设备
mci=mciGetDeviceID( open.lpstrDeviceType );        //获取该设备的 ID
mciSendCommand(mci,MCI_SET,MCI_WAIT|MCI_SET_DOOR_OPEN,NULL);
                                                   //打开光驱门
...                                                //省略部分代码
mciSendCommand(mci,MCI_SET,MCI_WAIT|MCI_SET_DOOR_CLOSED,NULL);
                                                   //关闭光驱门
```

在上面的代码中，用户首先应该初始化 CD 设备，然后再打开或关闭设备。用户可以在随书光盘中运行该实例程序看看效果。

用户也可以使用这两个结构体打开 MP3 文件并播放。代码如下：

```
    ...                                            //省略部分代码
    MCI_OPEN_PARMS open;                           //定义结构体变量
    open.lpstrElementName="C:\\oo.mp3";            //指定 MP3 文件路径
    open.lpstrDeviceType="mpegaudio";              //指定播放设备类型
    UINT err;                                      //定义错误变量
    err=mciSendCommand(NULL,MCI OPEN,MCI OPEN TYPE|MCI OPEN ELEMENT,(DWO
    RD)&open);                                     //初始化设备
    if(err==0)                                     //设备初始化成功
    {
    MCI_PLAY_PARMS play;                           //定义结构体变量
play.dwFrom=0;                                     //指定文件的播放位置
    mciSendCommand(open.wDeviceID,MCI_PLAY,0,(DWORD)&play);
                                                   //播放 MP3 文件
    }
    else                                           //初始化设备失败
    {
    char str[100];                                 //定义并初始化字符数组
    mciGetErrorString(err,(LPSTR)str,100);         //获取初始化设备时的错误信息
    MessageBox(str);                               //错误提示
    }
```

注意：如果函数 mciSendCommand()执行失败，则会返回错误信息。用户可以调用函数 mciGetErrorString()获取错误信息。如果用户使用上面的代码初始化音频设备失败或者播放 MP3 音乐无声音，则用户应该重新安装或更新声卡驱动程序。

在本小节中，主要向用户介绍了在 Windows 系统下 MP3 文件的基本格式和使用函数播放 MP3 文件的基本方法。请用户将随书光盘中的实例代码结合本书理论知识一起学习，这样将更有效率。

9.2　界 面 设 计

对于音乐播放器而言，界面中各个控件的位置以及大小是否合理决定了播放器设计是否成功，所以界面设计是非常重要的一关。在本章中，将在 VC 开发环境中进行界面的设计以及代码编写工作。用户设计播放器，应当首先创建其窗口。本节将向用户介绍在 VC 开发环境下，创建播放器工程以及播放器界面窗口。

9.2.1　创建工程

用户需要在 VC 中新建一个工程，并将其工程命名为“实用播放器”。具体创建步骤如下图所示：

（1）选择“文件”|“新建”命令，打开“新建”对话框，如图 9.3 所示。

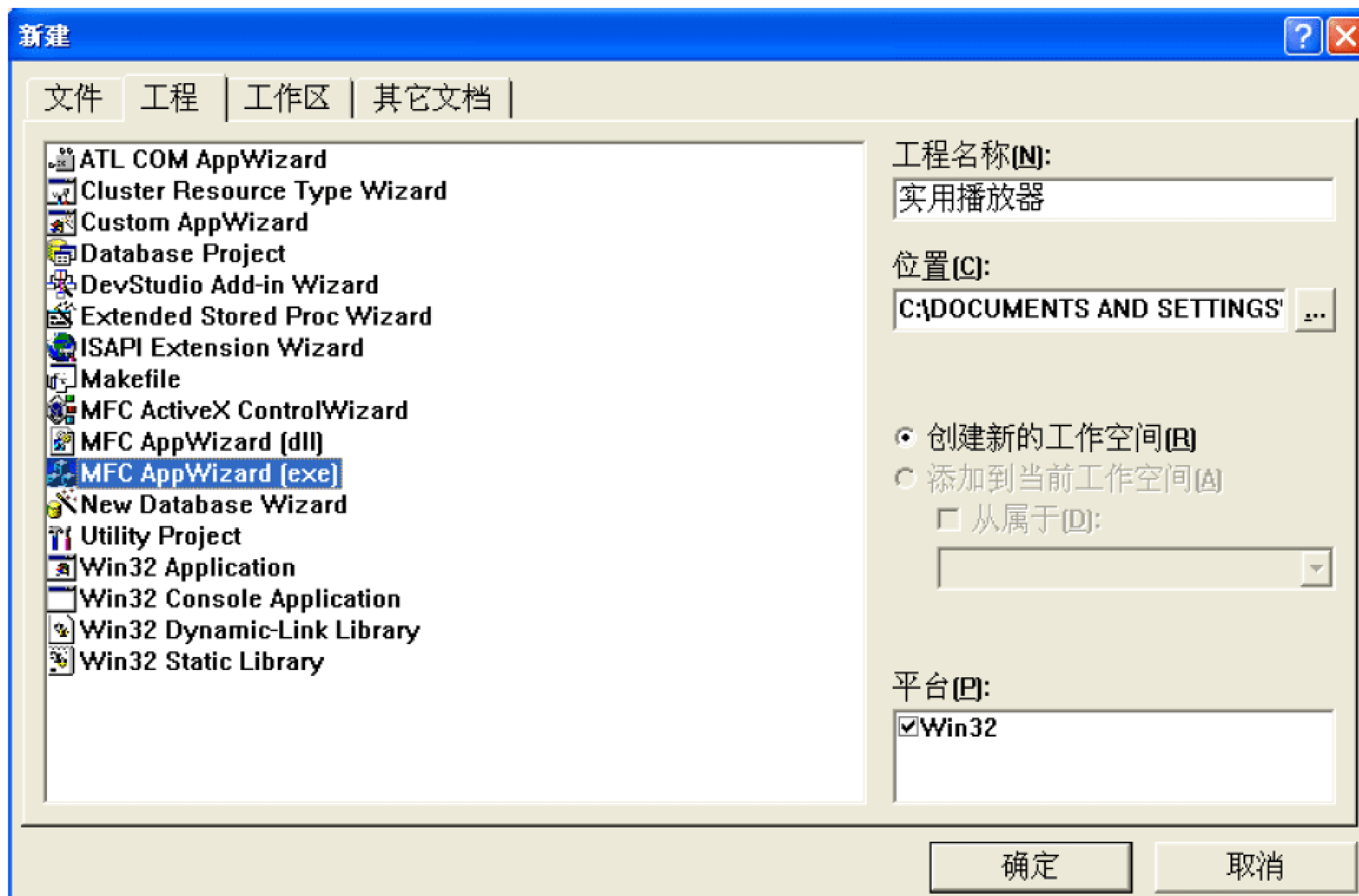


图 9.3　“新建”对话框

用户在该对话框中，选择新建基于 MF 应用程序向导的工程，并且可以修改工程名称以及工程的保存路径。

（2）单击“确定”按钮，进入下一步修改应用程序的类型，如图 9.4 所示。

由于该应用程序是基于对话框模式，所以用户在选择应用程序类型时，应该选择“基本对话框”单选按钮。

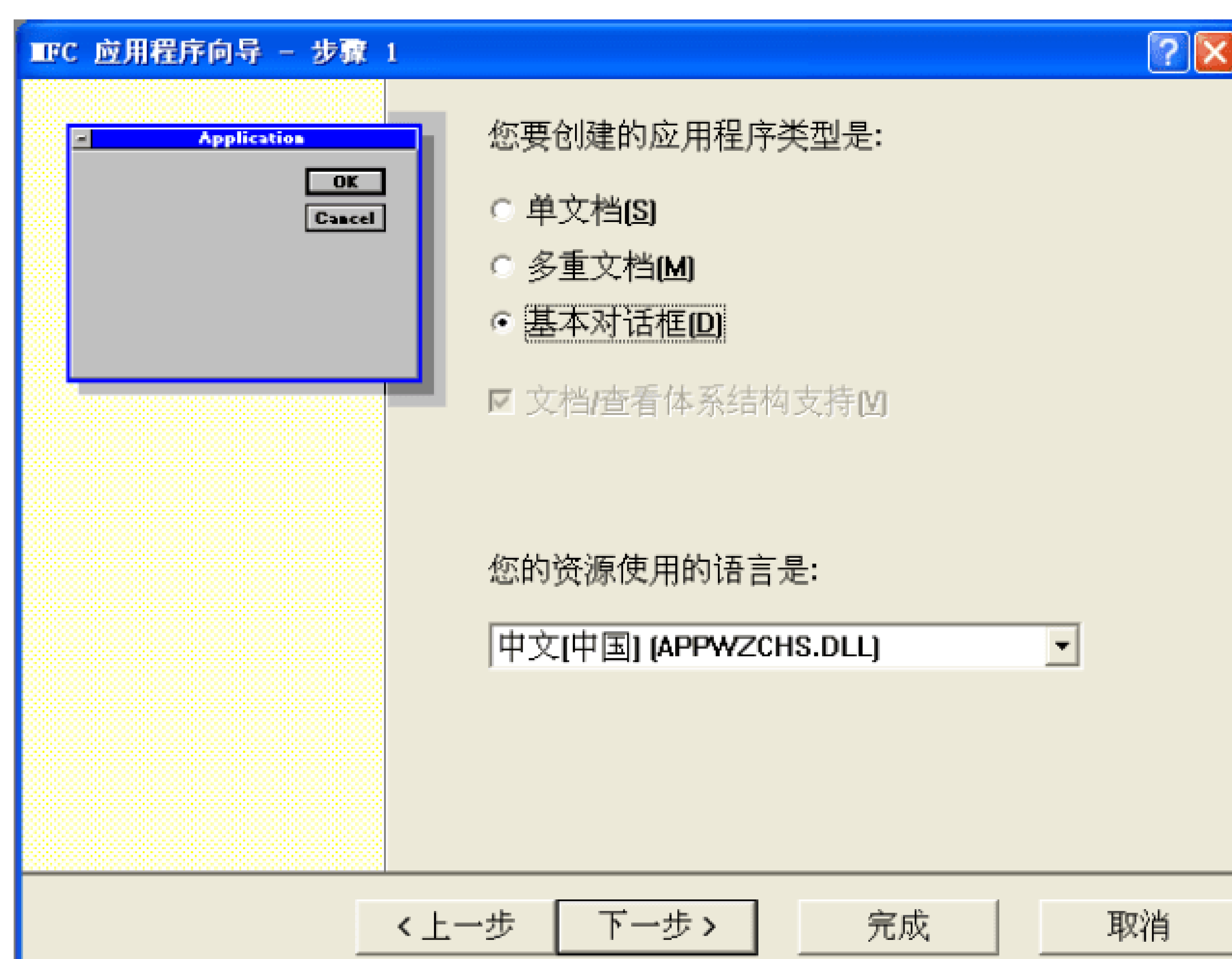


图 9.4　选择应用程序类型

（3）单击“下一步”按钮，进入设置步骤的第二步。由于该实例播放器不应该含有关于播放器对话框的任何信息。所以，在这里应该取消选中“"关于"对话框”复选框，如图 9.5 所示。

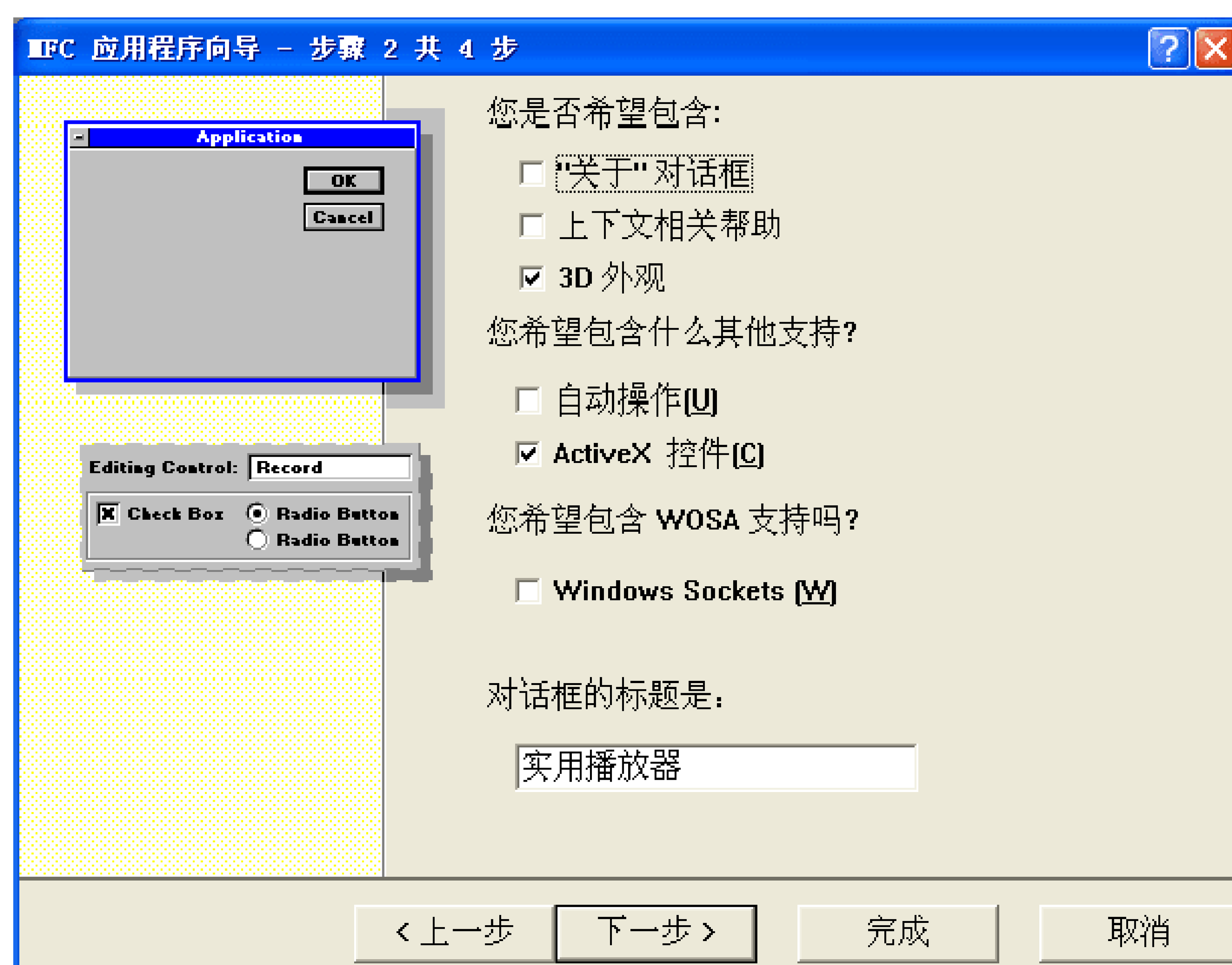


图 9.5　去掉“"关于"对话框”选项

（4）单击“完成”按钮，完成播放器工程的相关设置。

注意：由于本实例程序不包含套接字相关功能，所以在设置中并未选择工程支持套接字选项。

9.2.2　设计窗口

用户在播放器工程的资源管理器中，可以看到 VC 已经为该工程创建了一个默认的对

话框，如图 9.6 所示。但是用户可以根据需要修改该对话框的界面。

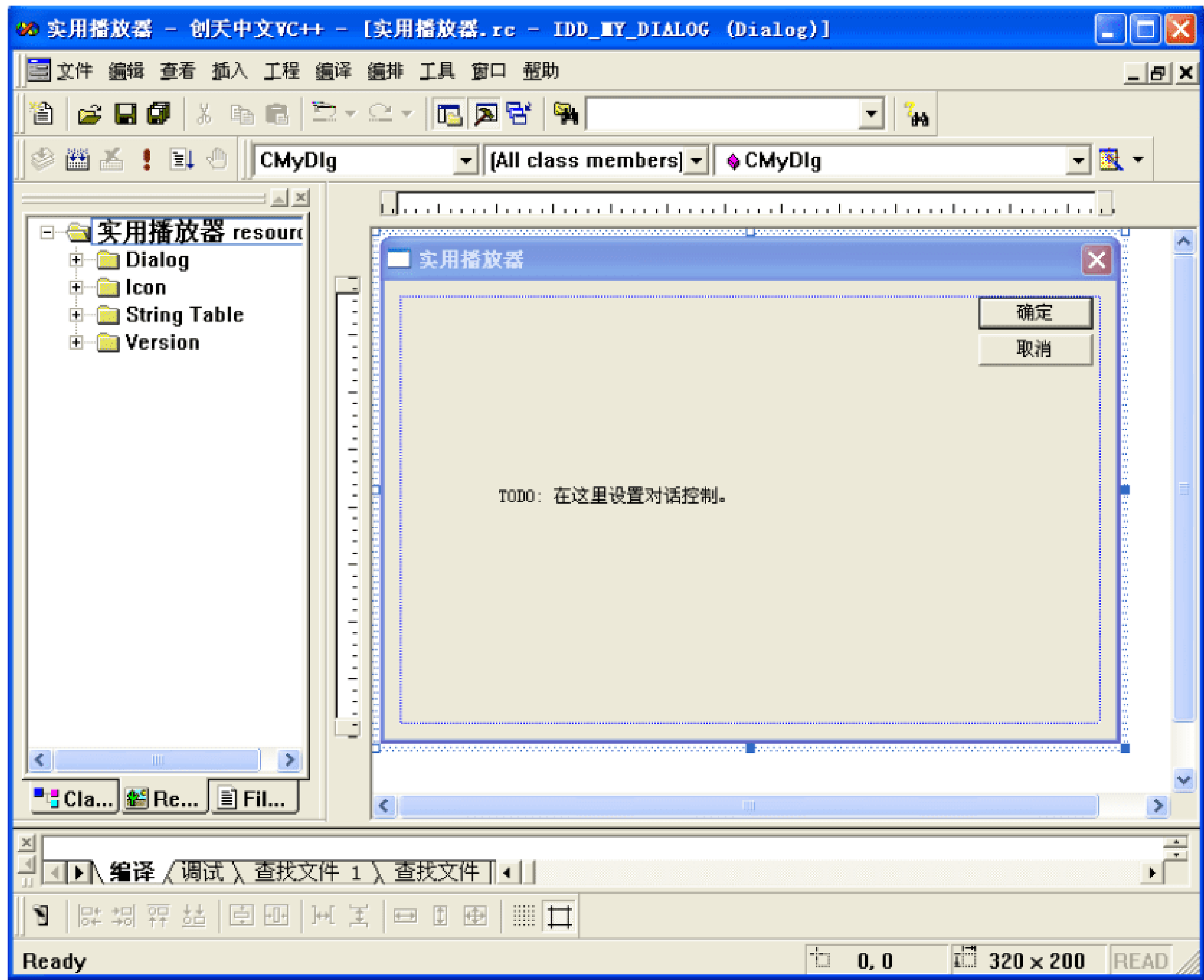


图 9.6　默认对话框

用户可以用鼠标将控件拖动到对话框面板上，并且调整各控件的大小以及位置。调整后的界面效果如图 9.7 所示。



图 9.7　调整后的播放器界面效果

用户在进行播放器界面设计时，使用了很多子窗口控件。关于各个控件的 ID、属性以及作用，如表 9.4 所示。

表 9.4　相关控件信息

控　件　ID	控 件 属 性	控 件 作 用
IDC_TUPIAN	图片控件	定时显示图片
IDC_PLAY	按钮控件	播放音乐
控件ID	控件属性	控件作用
IDC_ZANTING	按钮控件	暂停播放音乐
IDC_STOP	按钮控件	停止播放音乐
IDC_PRE	按钮控件	播放上一首音乐
IDC_NEXT	按钮控件	播放下一首音乐
IDC_ADD	按钮控件	添加音乐
IDC_PROGRESS1	进度条控件	显示音乐播放进度
IDC_TAB1	TAB控件	显示音乐列表

用户通过本节中的内容，应该了解了在 VC 中通过资源管理器设计实例程序界面的方法。关于各个控件的具体使用方法将在 9.3 节中讲解。

9.3　界面初始化

用户 VC 开发环境下编程，可以调用 MFC 函数实现窗口或者窗口中各个控件的初始化状态以及实现方法。因此，在本节中将向用户详细讲解播放器窗口界面中各个控件的初始化编程。

9.3.1　控件初始化

在 9.2 节中，用户通过 VC 资源管理器设计了播放器的界面，保存项目并且运行，程序运行后的界面如图 9.8 所示。播放器运行之后，用户可以看到界面上的控件都没有实现其应有的功能。这是因为，用户并没有为各个控件编写相应的功能代码。所以，在本节中，将向用户介绍播放器窗口中各个控件的初始化状态编程。

在程序初始化时，为了避免用户的误操作而造成程序发生异常。所以，关于播放控制的按钮都必须处于禁用状态，如图 9.9 所示。

用户禁用播放控制按钮功能，应该是在程序初始化函数 CMyDlg::OnInitDialog()中实现。在 VC 中实现该功能可以调用 MFC 函数库中的函数 GetDlgItem()获取指定 ID 控件的指针，然后使用该指针调用函数 EnableWindow()将控件窗口禁用。代码如下：

```
BOOL CMyDlg::OnInitDialog()
{
    CDialog::OnInitDialog();
    SetIcon(m_hIcon, TRUE);
    SetIcon(m_hIcon, FALSE);
```



图 9.8　播放器运行界面

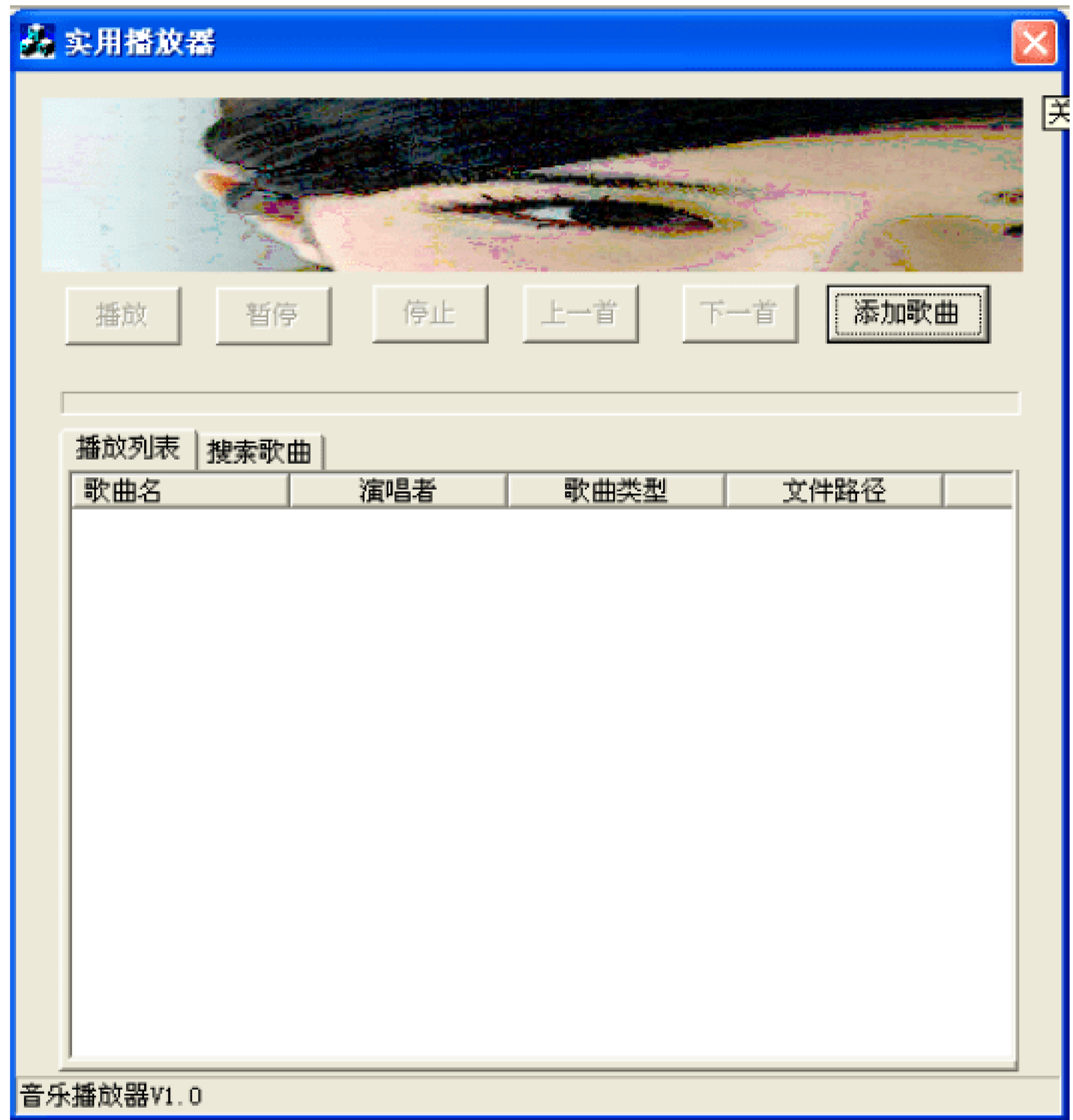


图 9.9　禁用播放控制按钮

```
  GetDlgItem(IDC_PLAY)->EnableWindow(false);        //禁用各个播放控制按钮
GetDlgItem(IDC_ZANTING)->EnableWindow(false);
GetDlgItem(IDC_PRE)->EnableWindow(false);
GetDlgItem(IDC_NEXT)->EnableWindow(false);
GetDlgItem(IDC_STOP)->EnableWindow(false);
...                                                 //省略部分代码
    return TRUE;
}
```

9.3.2　图片控件初始化

在程序启动时为了美化界面，用户还应该在界面窗口的图片控件中显示程序初始化图片。在控件中显示位图将使用到函数 StretchBlt()，该函数原型如下：

```
BOOL StretchBlt(
  HDC hdcDest,
  int nXOriginDest,
  int nYOriginDest,
  int nWidthDest,
  int nHeightDest,
  HDC hdcSrc,
  int nXOriginSrc,
  int nYOriginSrc,
  int nWidthSrc,
  int nHeightSrc,
  DWORD dwRop
);
```

该函数的作用是将兼容设备 DC 中的位图按照目标 DC 的实际大小进行粘贴。该函数若调用成功，则返回 true。否则，将返回 false。部分参数如下：

- 参数 hdcDesthdcDest 表示目标 DC 的句柄。
- 参数 nXOriginDest、nYOriginDest、nWidthDest、nHeightDest 需要组合使用，表示目标 DC 中被用来显示位图的区域大小。
- 参数 hdcSrc 表示存放位图的兼容 DC 句柄。
- 参数 nXOriginSrc、nYOriginSrc、nWidthSrc、nHeightSrc 需要组合使用，用于指定兼容 DC 中将被显示的位图区域。
- 参数 dwRop 表示位图显示方式。一般情况下，该值设置为 SRCCOPY，表示位图将从兼容 DC 中被直接复制到目标 DC 中。如果用户需要使用该参数的其他取值，可以参考 MSDN。

例如，在该实例程序界面中随机显示一幅位图，位图资源 ID 为 IDB_BITMAP1。代码如下：

```
...                                             //省略部分代码
bit=::LoadBitmap(AfxGetApp()->m_hInstance,MAKEINTRESOURCE(IDB_BITMAP1));
                                                //读取位图资源并返回其句柄
    dc1=::CreateCompatibleDC(::GetDC(::GetDlgItem(this->m_hWnd,IDC_TUPIA
    N)));
                                                //创建与位图控件相兼容的设备 DC
    ::SelectObject(dc1,bit);                    //将位图资源句柄选入设备兼容 DC 中
::StretchBlt(::GetDC(::GetDlgItem(this->m_hWnd,IDC_TUPIAN)),1,1,450,80,
dc1,0,0,400,330,SRCCOPY);
                                                //调用 API 函数将兼容 DC 中的位图复制到目
标 DC 中
...                                             //省略部分代码
```

首先，用户读取位图资源到应用程序中，再创建与目标 DC 的兼容 DC 并返回其句柄。

然后使用函数 SelectObject()将读取的位图选入到兼容设备 DC 中，再调用函数 StretchBlt()将兼容 DC 中的位图直接复制到目标 DC 中。

如果用户为了使该位图在界面中一直进行显示并且直到程序窗口关闭，那么用户只能将实现该功能的代码添加到函数 CMyDlg::OnPaint()中。因为，在窗口发生重绘时，程序都会调用函数 OnPaint()实现界面重绘工作。代码如下：

```
void CMyDlg::OnPaint()
{
    if (IsIconic())
    {
        CPaintDC dc(this);                 //定义设备上下文对象
        SendMessage(WM_ICONERASEBKGND, (WPARAM) dc.GetSafeHdc(), 0);
        ...                                //省略部分代码
        CRect rect;
        GetClientRect(&rect);
        int x = (rect.Width() - cxIcon + 1) / 2;
        int y = (rect.Height() - cyIcon + 1) / 2;
        dc.DrawIcon(x, y, m_hIcon);        //重绘程序窗口图标
    }
    else
    {
        CDialog::OnPaint();                //调用其基类的相应函数
    }
    ::SelectObject(dc1,bit);               //将位图资源句柄选入设备兼容 DC 中
::StretchBlt(::GetDC(::GetDlgItem(this->m_hWnd,IDC_TUPIAN)),1,1,450,80,
dc1,0,0,400,330,SRCCOPY);
                                //调用 API 函数将兼容 DC 中的位图复制到目标 DC 中
}
```

在程序中，用户在函数 SelectObject()中，使用了两个句柄变量 dc1 和 bit，分别表示兼容设备 DC 句柄和位图资源句柄。函数 SelectObject()的作用是将位图资源句柄 bit 选入到兼容设备 DC 中。如果用户在该函数中使用句柄 dc1 和 bit，则需要在窗口类 CMyDlg 中进行声明，并且在窗口初始化函数中进行初始化。代码如下：

```
class CMyDlg : public CDialog                    //窗口类 CMyDlg
{
public:
    CMyDlg(CWnd* pParent = NULL);
HBITMAP bit;                                     //位图句柄
HDC dc1;                                         //兼容 DC
 ...                                             //省略部分代码
  }
BOOL CMyDlg::OnInitDialog()                      //窗口初始化函数
{
    CDialog::OnInitDialog();
    SetIcon(m_hIcon, TRUE);
    SetIcon(m_hIcon, FALSE);
GetDlgItem(IDC_PLAY)->EnableWindow(false);       //禁用控件
GetDlgItem(IDC_ZANTING)->EnableWindow(false);
GetDlgItem(IDC_PRE)->EnableWindow(false);
GetDlgItem(IDC_NEXT)->EnableWindow(false);
GetDlgItem(IDC_STOP)->EnableWindow(false);
    bit=::LoadBitmap(AfxGetApp()->m_hInstance,MAKEINTRESOURCE(IDB_BITMAP
    1));
```

```
                                    //读取位图资源并返回其句柄
    dc1=::CreateCompatibleDC(::GetDC(::GetDlgItem(this->m hWnd,IDC TUPIA
    N)));
                                    //创建与位图控件相兼容的设备 DC
    ...                             //省略部分代码
    return TRUE;
}
```

代码中，API 函数 LoadBitmap()的作用是读取位图资源 IDB_BITMAP1 到应用程序中，并返回其位图句柄。然后，使用函数 CreateCompatibleDC()创建与位图控件相兼容的设备 DC 并返回其句柄。保存程序并运行，如图 9.10 所示。



图 9.10　程序初始化运行界面

9.3.3　TAB 控件初始化

如果程序启动后，其所在目录中没有响应歌曲，则在 TAB 控件中提示用户“播放器中没有任何歌曲”需要添加歌曲。否则，播放歌曲。首先，使用快捷键 Ctrl+W 打开应用程序向导的 Member Variables 选项卡，为 TAB 控件添加相关变量，如图 9.11 所示。然后，在 ID 列表中选择 IDC_TAB1 选项，再单击 Add Variables 按钮添加该 ID 控件对象，如图 9.12 所示。

用户可以通过添加成员变量对话框修改变量名称为 m_tab。使用 CTabCtrl 类对象 m_tab 在 TAB 控件中添加属性页，实现该功能的函数是 InsertItem()，其原型如下：

```
BOOL InsertItem( UINT nMask, int nItem, LPCTSTR lpszItem, int nImage, LPARAM
lParam );
```

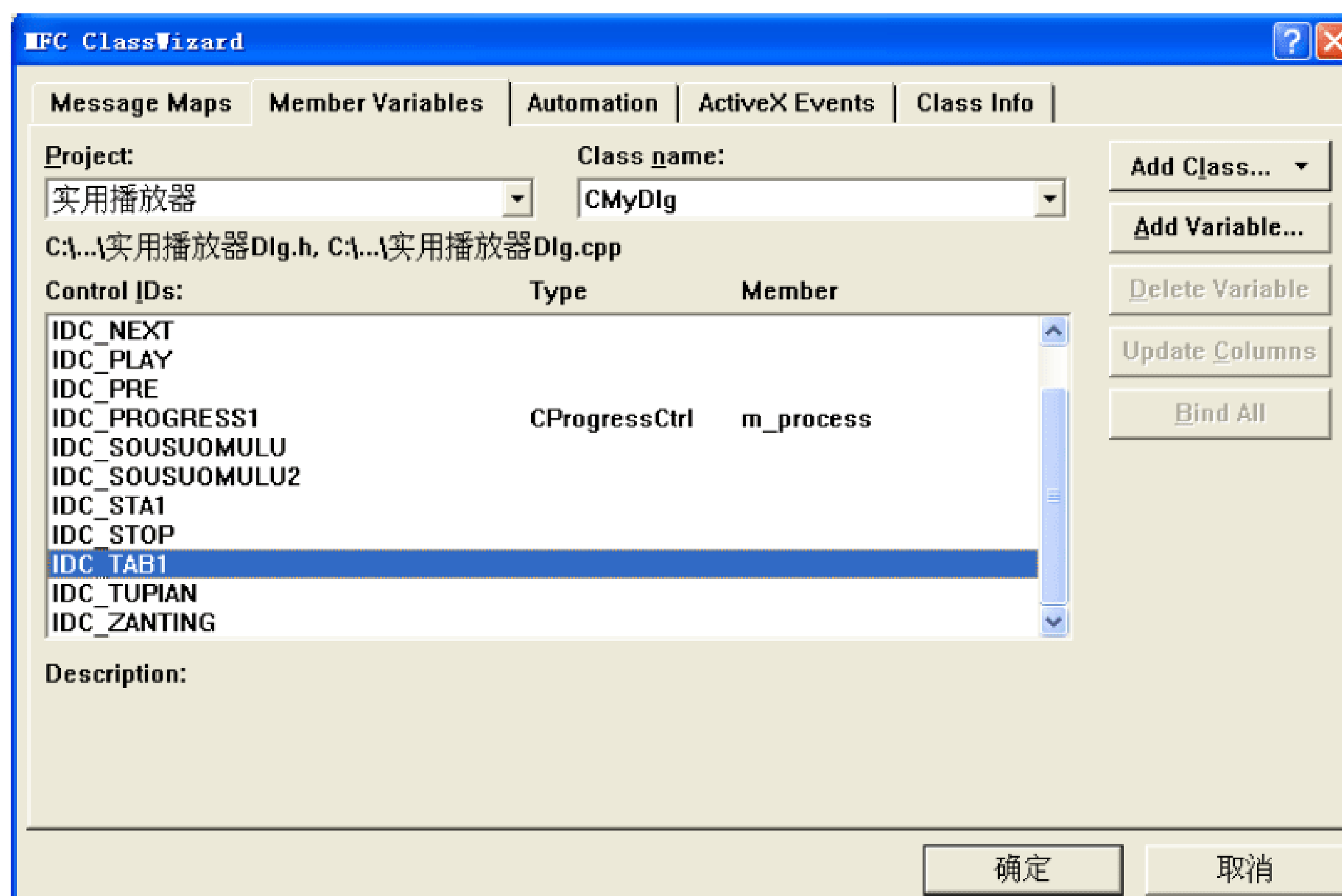


图 9.11　应用程序向导的 Member Variables 选项卡

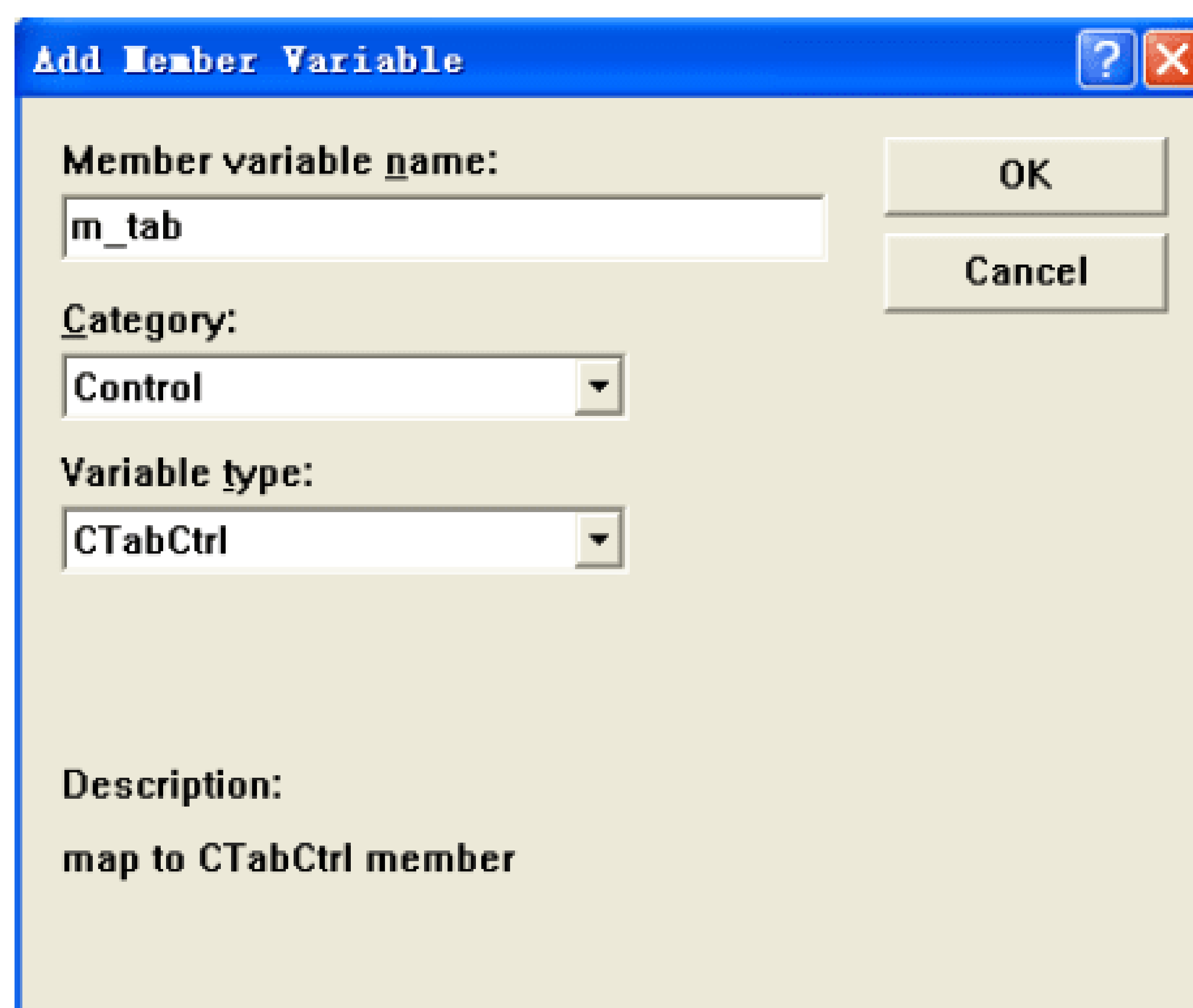


图 9.12　添加 TAB 控件相关变量

该函数的作用是在 TAB 控件中插入属性页。部分参数如下：

- ❑ 参数 nMask 表示属性页状态标志，在本章中指定为 TCIF_TEXT，表示该属性页的标题文字有效。
- ❑ 参数 nItem 表示将操作的属性页序号。
- ❑ 参数 lpszItem 表示属性页标题。

该函数的其他参数均可以设置为 NULL。例如，在程序中调用该函数向 TAB 控件中插入“播放列表”以及“搜索歌曲”两个属性页。代码如下：

```
BOOL CMyDlg::OnInitDialog()
{
    CDialog::OnInitDialog();
   ...                                                    //省略部分代码
   m_tab.InsertItem(TCIF_TEXT,0,"播放列表",NULL,NULL);
```

```
                                                        //添加播放列表属性页
m_tab.InsertItem(TCIF_TEXT,1,"搜索歌曲",NULL,NULL);      //添加搜索歌曲属性页
}
```

程序中调用函数 CTabCtrl::InsertItem()向 TAB 控件中添加两个属性页，分别是播放列表和搜索歌曲。运行代码，用户可以在界面中看到这两个属性页，如图 9.13 所示。



图 9.13　添加属性页

如果用户在 TAB 控件中添加属性页成功，便可以向该控件中添加列表控件和按钮控件实现各个属性页界面。界面中添加的控件，如表 9.5 所示。

表 9.5　界面中的控件

控　件　ID	控 件 类 型	控 件 作 用
IDC_LIST2	列表控件	显示歌曲列表
IDC_SOUSUOMULU	按钮控件	搜索本目录下的歌曲
IDC_SOUSUOMULU2	按钮控件	搜索本地所有歌曲

用户将表 9.5 中所示的控件添加到界面中后，需要根据属性页的不同而选择性的显示这些控件。例如，当程序显示属性页“播放列表”时，只能有列表控件显示，其他控件则隐藏。在 MFC 中，如果 TAB 属性页状态发生改变，那么系统会发送消息 TCN_SELCHANGE 到其父窗口中。用户可以响应该消息进行判断属性页的当前状态。根据用户当前所查看的属性页，调整各个控件的显示状态。代码如下：

```
void CMyDlg::OnSelchangeTab1(NMHDR* pNMHDR, LRESULT* pResult)
{
int n=m_tab.GetCurSel();                        //返回当前属性页的序号
switch(n)
```

```
{
case 0:                                                 //显示播放列表属性页
{
    GetDlgItem(IDC_SOUSUOMULU)->ShowWindow(false);      //隐藏按钮控件
    GetDlgItem(IDC_SOUSUOMULU2)->ShowWindow(false);
    GetDlgItem(IDC_LIST2)->ShowWindow(true);            //显示列表控件
    break;
}
case 1:                                                 //显示搜索歌曲属性页
    {
        GetDlgItem(IDC_SOUSUOMULU)->ShowWindow(true);   //显示搜索目录按钮
        GetDlgItem(IDC_SOUSUOMULU2)->ShowWindow(true);
        GetDlgItem(IDC_LIST2)->ShowWindow(false);       //隐藏列表控件
        break;
    }
}
    *pResult = 0;
}
```

在程序中，用户调用函数 CTabCtrl::GetCurSel()返回当前显示的属性页序号。然后根据该序号设置各个控件的显示或者隐藏状态，如图 9.14 和图 9.15 所示。



图 9.14　显示播放列表

用户在程序中实现各个控件的显示状态以后，本章有关 TAB 控件初始化的工作就基本完成了。请用户在学习过程中，可以试着在实例程序的基础上添加一些 TAB 控件的其他功能。这样，有助于用户加深理解 TAB 控件的用法。



图 9.15　显示搜索属性页

用户在上一节中向 TAB 属性页中添加了列表控件，该列表用于显示已经存在的歌曲。在本节中，将向用户介绍列表控件的初始化。首先，利用应用程序向导为列表控件关联 CListctrl 对象，并将对象名称设置为 m_list，如图 9.16 所示。

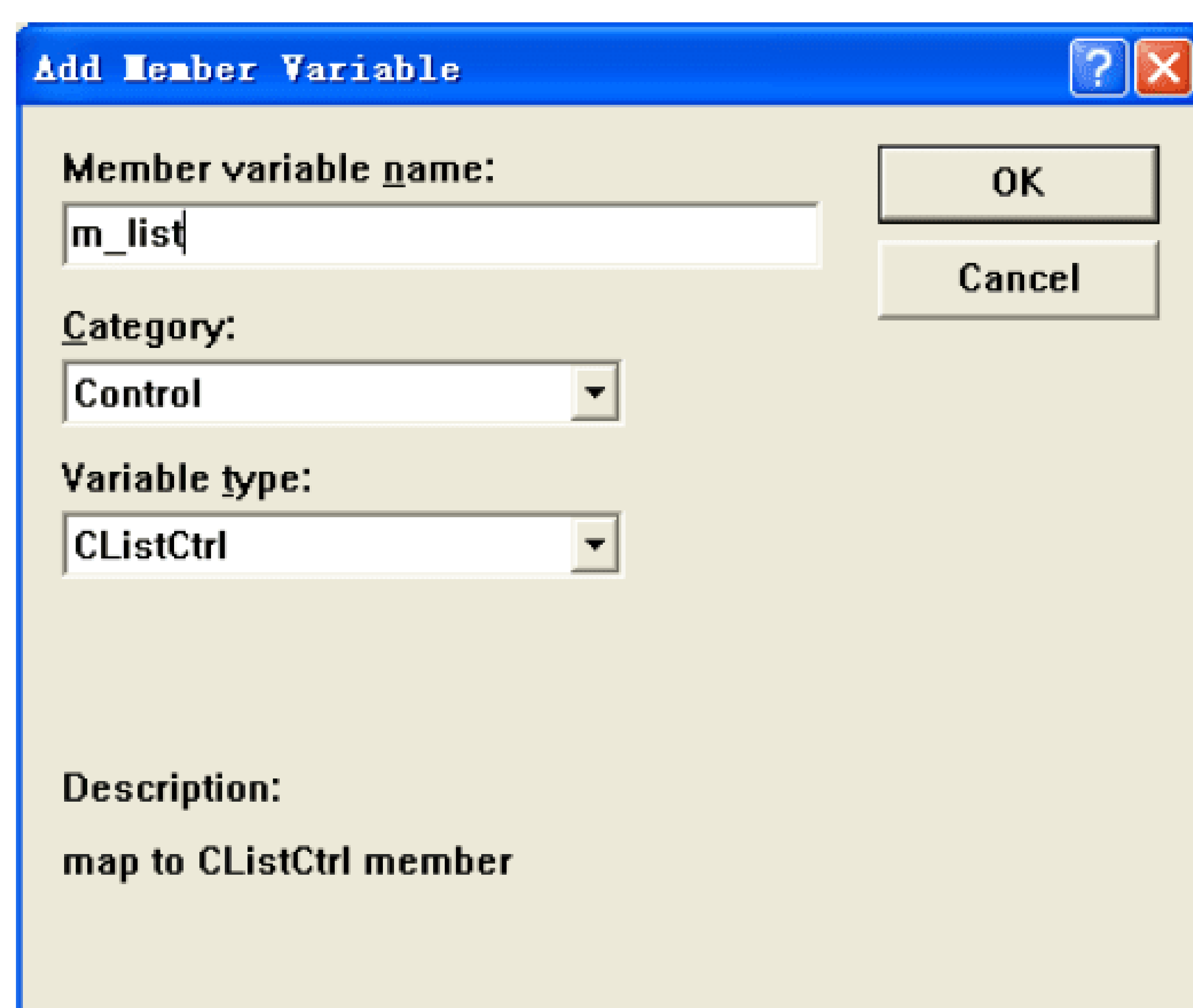


图 9.16　添加列表控件对象

然后，在对话框初始化函数 OnInitDialog()中，调用 CListctrl 类相关函数对列表控件进行初始化。代码如下：

```
BOOL CMyDlg::OnInitDialog()
{
    CDialog::OnInitDialog();
```

```
    ...                                         //省略部分代码
LVCOLUMN lv;                                    //定义 LVCOLUMN 结构变量
lv.mask=LVCF_TEXT|LVCF_FMT|LVCF_WIDTH;          //填充 LVCOLUMN 结构体
lv.fmt=LVCFMT_CENTER;
lv.pszText="歌曲名";
lv.cx=100;                                      //设置项目宽度
m_list.InsertColumn(0,&lv);                     //插入项目
lv.pszText="歌曲长度";
m_list.InsertColumn(1,&lv);
lv.pszText="歌曲类型";
m_list.InsertColumn(2,&lv);
}
```

用户将以上代码编译运行，将看到一个具有项目标题的列表控件，如图 9.17 所示。



图 9.17　初始化后的列表控件

在本节中，主要向用户介绍了音乐播放器界面初始化的方法以及各个控件的使用等。通过本节的学习，用户应该掌握 TAB 控件与列表控件的基本使用方法。

9.3.4　进度条、状态栏

在程序中，进度条的作用是指示歌曲的播放位置以及歌曲搜索的进度等。而状态栏的作用是显示一些提示信息给用户或者播放器的使用方法等。

1．进度条

在 MFC 中，进度条控件类是 CProgressCtrl，本节将使用该控件类的相关函数实现进度条的各个功能。在程序中，初始化进度条长度的函数是 CProgressCtrl::SetRange32()。其

原型如下：

```
void SetRange32( int nLower, int nUpper );
```

该函数用于指定进度条的作用范围。其参数分别表示最小值与最大值。例如，将进度条的作用范围设置为 0～100。代码如下：

```
m_process. SetRange32( 0,100 );          //设置进度条作用范围为 0～100
```

注意：进度条的作用范围最小值不能小于 0。

获取或设置进度条当前位置的函数分别是 GetPos()和 SetPos()。其原型如下：

```
int GetPos( );                           //获取进度条当前位置
int SetPos( int nPos );                  //设置进度条当前位置
```

函数 GetPos()调用成功，将返回进度条的当前位置。函数 SetPos()用于设置进度条的当前位置，其参数 nPos 表示将设置的位置索引。例如，使进度条的当前位置前进 10 个单位索引。代码如下：

```
...                                      //省略部分代码
m_process. SetRange32( 0,100 );          //设置进度条作用范围为 0～100
int i=m_process.GetPos();                //获得当前位置索引
m_process.SetPos(i+10);                  //设置当前位置索引为 i+10
...                                      //省略部分代码
```

设置进度条位置的步进变化，可以使用函数 SetStep()和 StepIt()实现。其原型如下：

```
int SetStep( int nStep );
int StepIt();
```

函数 SetStep()调用成功，将返回进度条变化前的位置索引。其参数 nStep 指定进度条变化的单位量。函数 StepIt 是执行步进操作。例如，使进度条每次前进 5 个单位。代码如下：

```
...                                      //省略部分代码
m_process. SetRange32( 0,100 );          //设置进度条作用范围为 0～100
  int i=m_process.GetPos();              //获得当前位置索引
  m_process.SetPos(i+10);                //设置当前位置索引为 i+10
  while(m_process.GetPos()<=100)         //如果进度条当前位置小于  100
   {
     m process. SetStep(5);              //进度条每次以 5 个单位进行变化
       m_process.StepIt();               //执行步进操作
       }
  ...                                    //省略部分代码
```

在代码中，进度条以 5 个单位进行位置变化，如图 9.18 所示。

在本实例程序中，进度条主要被用于指示歌曲的播放进度。所以，只需要将进度条的范围设置为音乐文件的长度即可。然后，根据已读取的文件长度占总长度的大小设置进度条的当前位置。



图 9.18　变化中的进度条

2. 状态栏

一般，用户在程序中需要使用状态栏显示一些该程序的相关帮助信息。用户在程序中创建状态栏应该首先在主对话框类 CMyDlg 中定义一个状态栏句柄对象 statu。然后，在对话框初始化函数 OnInitDialog()中，使用函数 API 函数创建状态栏并返回其句柄。该函数原型如下：

```
HWND CreateStatusWindow(
    LONG style,
    LPCTSTR lpszText,
    HWND hwndParent,
    UINT wID
);
```

该函数的作用是创建状态栏对象并返回其句柄。该函数具有 4 个参数，意义分别如下：

- 参数 style 表示状态栏创建时所指定的窗口样式。其取值一般可以设置为 WS_CHILD|WS_VISIBLE，表示用户所创建窗口的子窗口并且是显示的。如果用户还需要为状态栏指定其他窗口样式，则可以参考 MSDN。
- 参数 lpszText 表示用户将在状态栏上显示的文字信息。
- 参数 hwndParent 表示状态栏的父窗口句柄。用户在程序中可以使用 this 指针获取该句柄。例如，this->m_hWnd 或 this->GetParent()->m_hWnd。

注意：上面所讲的 this 指针在程序中，表示对象本身。

- 参数 wID 指定状态栏的窗口 ID。用户指定的该 ID 值必须是在程序中定义的 ID 值。例如，用户定义的 ID 值是 IDC_123，代码如下：

```
#define IDC_123                        3251
```

如果用户在窗口中创建一个状态栏用于显示相关消息，则应该首先在头文件“实用播

放器 Dlg.h”中定义一个状态栏句柄。代码如下：

```
class CMyDlg : public CDialog              //主窗口类定义
{
public:
    CMyDlg(CWnd* pParent = NULL);
       HWND statu;                         //定义状态栏句柄
...                                        //省略部分代码
   }
```

然后，用户在窗口初始化函数中调用函数 CreateStatusWindow()创建状态栏并为其指定窗口 ID。代码如下：

```
BOOL CMyDlg::OnInitDialog()                //窗口初始化函数
{
    CDialog::OnInitDialog();
statu=::CreateStatusWindow(WS_CHILD|WS_VISIBLE,"音乐播放器 V1.0",this->
m_hWnd,IDC_123);
                                           //创建状态栏
...                                        //省略部分代码
    }
```

用户在程序中添加以上代码后，编译运行程序，效果如图 9.19 所示。



图 9.19　成功创建状态栏

如果用户在程序编写过程中需要在状态栏上显示信息，则可以直接使用状态栏句柄 statu 即可。例如，用户在状态栏内显示系统当前时间。代码如下：

```
void CMyDlg::OnTimer(UINT nIDEvent)
{
  CTime time;                                           //定义时间类对象
time.GetCurrentTime();                                  //获取系统当前时间
  CString str=time.Format("当前时间: % H: % M:%S"); //格式化字符串
  ::SetWindowText(statu,str);                           //设置状态栏显示的文字
  }
```

上面的程序运行之后，会在状态栏上显示当前系统的时间，如图 9.20 所示。



图 9.20　在状态栏内显示当前时间

9.4　添加消息映射

用户在 VC 编译器中，可以使用 MFC 的消息映射功能为界面中的各个子控件添加相应的功能响应函数。例如，用户对按钮控件的单击操作等。在控件的消息响应函数中，用户通过编写代码以实现真正意义上的控件功能。在本节中，将向用户具体介绍 MFC 消息映射表的功能以及在程序中手动添加消息映射宏的方法。

9.4.1　MFC 消息映射表

在 MFC 中，消息映射表的作用是将用户对控件的一些操作消息与该消息的响应函数相关联。但是，由于消息映射表的结构定义很复杂且用户使用起来不方便。所以，在 MFC 中使用消息映射宏进行消息映射功能。例如，用户在 MFC 程序中，经常会看见编译器自动添加的一些消息映射宏代码。代码如下：

```
...                                              //省略部分代码
BEGIN_MESSAGE_MAP(CMy121Dlg, CDialog)            //开始消息映射
//{{AFX_MSG_MAP(CMy121Dlg)
ON_WM_PAINT()                                    //系统刷新消息映射
ON_MESSAGE(MCI_NOTIFY,ONnot)                     //自定义消息映射
//}}AFX_MSG_MAP
END_MESSAGE_MAP()                                //结束消息映射
```

在以上代码中，消息是与消息响应函数通过消息映射宏联系起来并成对出现。例如，当窗口程序接收到消息 MCI_NOTIFY 后，程序将调用与该消息相对应的响应函数 ONnot()

实现相应的功能。

注意：用户在消息映射宏中，可以通过向导程序添加一些系统定义消息。如果是用户自定义的消息，则用户只能在消息映射宏中手动添加消息映射代码。

用户在消息映射宏中，所使用的消息响应函数必须首先在窗口类中进行声明。例如，用户消息响应函数 ONnot()的声明。代码如下：

```
class CMy121Dlg : public CDialog
{
protected:
    //{{AFX_MSG(CMy121Dlg)
    virtual BOOL OnInitDialog();
    afx_msg void OnPaint();
    afx_msg void ONnot();                   //消息响应函数的声明
  //}}AFX_MSG
...                                         //省略部分代码
  }
```

如果用户使用的消息类型是自定义，那么还需要在类定义之外定义该消息。否则，程序运行时会发生错误。例如，用户自定义消息 MCI_NOTIFY，代码如下：

```
#define MCI_NOTIFY         WM_USER+100
```

以上几个步骤是用户响应一个自定义消息并实现其功能必须操作的步骤。

9.4.2　使用消息映射宏

在 MFC 中，由于消息映射表结构对于程序员而言较为复杂，所以，MFC 使用消息映射宏代替了消息映射表结构。

1．消息映射宏

用户在实际编程时，使用的消息映射宏是 DECLARE_MESSAGE_MAP、BEGIN_MESSAGE_MAP 以及 END_MESSAGE_MAP。部分定义代码分别如下：

```
#define DECLARE_MESSAGE_MAP() \             //定义消息映射宏 DEC-
LARE_MESSAGE_MAP
private: \
    static const AFX_MSGMAP_ENTRY _messageEntries[]; \
protected: \
    static AFX_DATA const AFX_MSGMAP messageMap; \
    static const AFX_MSGMAP* PASCAL _GetBaseMessageMap(); \
    virtual const AFX_MSGMAP* GetMessageMap() const; \
#define BEGIN_MESSAGE_MAP(theClass, baseClass) \
                                            //定义消息映射宏 BEGIN_MESSAGE_MAP
    ...                                     //省略部分代码
    AFX_COMDAT const AFX_MSGMAP_ENTRY theClass::_messageEntries[] = \
    { \
#define END_MESSAGE_MAP() \                 //定义消息映射宏 END_MESSAGE_MAP
        {0, 0, 0, 0, AfxSig_end, (AFX_PMSG)0 } \
    }; \
```

以上代码便是 MFC 消息映射宏的定义代码。在代码中，DECLARE_MESSAGE_MAP 包含了消息映射所必须的变量和函数方法。而 BEGIN_MESSAGE_MAP 则包含了这些变量

的初始化以及函数的实现方法。最后，END_MESSAGE_MAP 结束消息映射宏。

注意：在宏 BEGIN_MESSAGE_MAP 中有两个参数 theClass 和 baseClass，分别表示调用该宏的类名以及该类的父类名。

例如，用户希望一个新建类具有消息映射功能，则首先应该在该类的定义中添加宏 DECLARE_MESSAGE_MAP()，表示该类支持消息映射。代码如下：

```
class CMyDlg : public CDialog
{
  ...                                        //具体的类定义
 DECLARE MESSAGE MAP()                       //表示该类支持消息映射功能
 }
```

然后，再在该类实现文件中添加消息映射宏标记。代码如下：

```
...                                          //具体的方法实现代码
BEGIN_MESSAGE_MAP(CMyDlg, CDialog)           //开始消息映射
    //{{AFX_MSG_MAP(CMyDlg)
    ON WM PAINT()                            //关联消息与消息响应函数
    ON_WM_QUERYDRAGICON()
    ON_NOTIFY(TCN_SELCHANGE, IDC_TAB1, OnSelchangeTab1)
    ON WM TIMER()
    ON BN CLICKED(IDC ADD, OnAdd)
    //}}AFX_MSG_MAP
END_MESSAGE_MAP()                            //结束消息映射
...                                          //省略部分代码
```

用户添加以上代码以后，新建类便具有了消息映射功能。如果用户手动添加消息响应代码，则只需要在宏 BEGIN_MESSAGE_MAP 和 END_MESSAGE_MAP 之间添加即可。

2. 添加消息响应函数

在本章实例中，由于子控件较多，所以将使用 MFC 向导程序为各子控件添加相应的消息响应函数。首先，用户在 VC 主界面中按下 Ctrl+W 组合键，弹出 MFC ClassWizard 对话框，如图 9.21 所示。

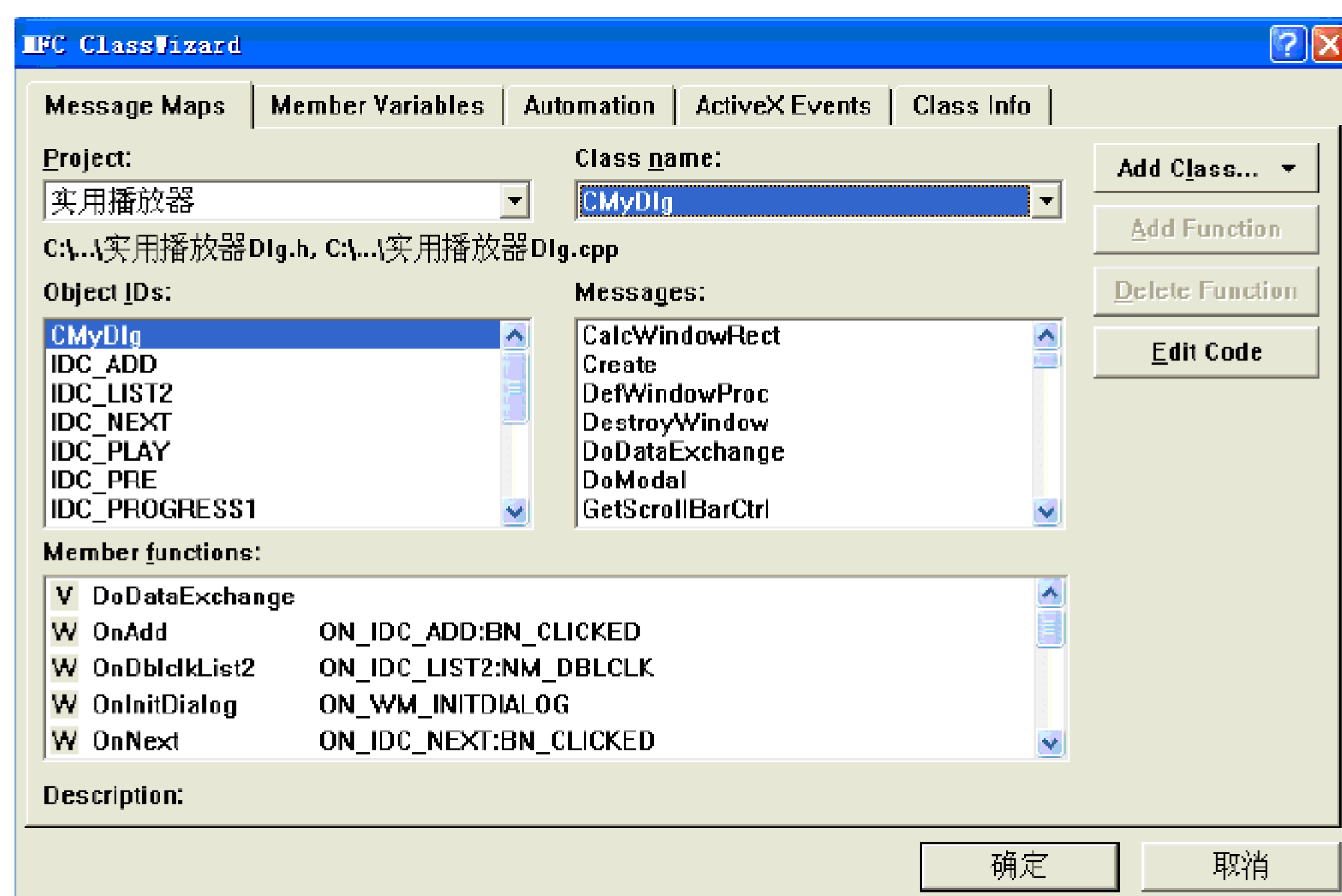


图 9.21　MFC ClassWizard 对话框

然后，用户可以在 Object ID 列表中找到相关控件 ID，再在 Messages 列表中选中将为其添加的系统消息，单击 Add Function 按钮即可。

用户使用该对话框将相应控件的消息响应函数添加完成之后，可以回到代码中查看消息映射宏的变化。代码如下：

```
...                                                  //省略部分代码
BEGIN_MESSAGE_MAP(CMyDlg, CDialog)                   //开始消息映射
    //{{AFX_MSG_MAP(CMyDlg)
    ON_WM_PAINT()
    ON_WM_QUERYDRAGICON()
    ON_NOTIFY(TCN_SELCHANGE, IDC_TAB1, OnSelchangeTab1)
    ON_WM_TIMER()
    ON_BN_CLICKED(IDC_ADD, OnAdd)                    //添加歌曲按钮的消息映射
    ON_BN_CLICKED(IDC_NEXT, OnNext)                  //下一首按钮的消息映射
    ON_BN_CLICKED(IDC_PLAY, OnPlay)                  //播放按钮的消息映射
    ON_BN_CLICKED(IDC_PRE, OnPre)                    //上一首按钮的消息映射
    ON_BN_CLICKED(IDC_STOP, OnStop)                  //停止按钮的消息映射
    ON_BN_CLICKED(IDC_SOUSUOMULU, OnSousuomulu)//搜索目录按钮的消息映射
    ON_BN_CLICKED(IDC_SOUSUOMULU2, OnSousuomulu2)
    ON_BN_CLICKED(IDC_ZANTING, OnZanting)            //暂停按钮的消息映射
    //}}AFX_MSG_MAP
END_MESSAGE_MAP()                                    //结束消息映射
...                                                  //类实现代码
```

用户通过以上代码可以发现，使用应用程序向导为控件添加消息响应函数和使用消息映射宏手动添加消息响应函数的效果是一样的。

注意：用户在本节中所添加的控件响应函数的实现方法，将在后面的小节中进行具体讲述。

9.5　多线程通信

在实例程序中，使用多线程不但可以使程序同时处理多个文件或实现多个功能，还可以利用多线程同步技术防止程序出现资源共享问题。在本节中，主要向用户介绍实例程序的线程分配以及线程间的通信。

9.5.1　线程分配

在本实例中，主要的线程分为播放线程和进度条设置线程。用户需要在播放线程中创建一个进度条设置线程，以便根据当前文件的播放状态设置进度条的当前位置。

首先，在播放按钮的消息响应函数中使用函数 mciSendCommand()打开音频设备并播放指定的 MP3 文件。代码如下：

```
void CMyDlg::OnPlay()                                //播放按钮消息响应函数
{
MCI_OPEN_PARMS open={0};                             //定义并初始化结构体
char str1[100];                                      //定义字符数组
```

```
open.lpstrElementName="F:\\音乐\\11\\齐秦-大约在冬季.wma";//指定播放文件路径
open.lpstrDeviceType="mpegvideo";                 //指定播放设备
DWORD err;                                        //定义错误信息
err=mciSendCommand(0,MCI_OPEN,MCI_OPEN_TYPE|MCI_OPEN_ELEMENT|MCI_WAIT,(
DWORD)(LPVOID)&open);                             //初始化音频设备
if(err==0)                                        //如果初始化设备成功
{
MCI_PLAY_PARMS play;                              //定义结构体变量
play.dwFrom=0;                                    //指定播放位置为起始位置
play.dwCallback=NULL;                             //返回消息的窗口句柄为 NULL
mciSendCommand(open.wDeviceID,MCI_PLAY,0,(DWORD)&play);
                                                  //播放指定文件
}
else                                              //初始化失败
{
    mciGetErrorString(err,(LPSTR)str1,100);       //获取失败信息
    MessageBox(str1);                             //显示失败信息
}
}
```

然后，在函数 OnPlay()中调用函数 CreateThread()创建进度条设置线程并将其启动。代码　如下：

```
void CMyDlg::OnPlay()                         //播放按钮消息响应函数
{
  ...                                         //省略部分代码
  HANDLE h;
  h=::CreateThread(NULL,0,setprocess,this->m_hWnd,0,NULL);
                                              //创建进度条设置线程
  ...                                         //省略部分代码
 }
```

在代码中，用户使用函数 CreateThread()创建了进度条设置线程并指定线程函数为 setprocess()。该函数在程序中必须声明为全局函数。否则，系统将创建线程失败。代码如下：

```
DWORD WINAPI setprocess(LPVOID lpParameter);      //声明线程函数
DWORD npos;                                       //定义全局变量用于保存播放位置
class CMyDlg : public CDialog
{
  ...                                             //省略部分代码
   }
```

现在，用户在播放 MP3 文件的同时，也启动了进度条设置线程。用户在该线程函数中可以根据当前文件的播放进度设置进度条的位置。代码如下：

```
DWORD WINAPI setprocess(LPVOID lpParameter)
{
CString str,str1;                                 //定义字符串变量
char ch[100];                                     //定义字符数组
MCI_STATUS_PARMS stat={0};                        //定义并初始化结构体变量
stat.dwItem=MCI_STATUS_LENGTH;                    //获取播放文件总长度
mciSendCommand(open.wDeviceID,MCI_STATUS,MCI_STATUS_ITEM,(DWORD)&stat);
                                                  //获取播放文件时间长度
npo=&stat.dwReturn;                               //返回播放长度值
BYTE a=MCI_MSF_MINUTE(npo);                       //获取时间长度的分钟值
```

```
BYTE b=MCI_MSF_SECOND(npo);                           //获取时间长度的秒值
npos=MCI_MAKE_HMS(0,a,b);                             //返回时间组合值
str.Format("%d",npos);
str+="/";                                             //添加符号"/"
while(1)
{
Sleep(1000);                                          //线程暂停 1 秒
stat.dwItem=MCI_STATUS_POSITION ;                     //获取当前播放状态标记
err=mciSendCommand(open.wDeviceID,MCI_STATUS,MCI_STATUS_ITEM,(DWORD)&st
at);
                                                      //获取文件的当前播放时间
str1.Format("%d",stat.dwReturn);                      //格式化字符串
strcat((char*)str.GetBuffer(0),(char*)str1.GetBuffer(0));
                                                      //连接字符串
::SetWindowText((HWND)lpParameter,str);               //设置控件标题
}
mciGetErrorString(err,(LPSTR)ch,100);                 //获取 MCI 错误信息
::SetWindowText((HWND)lpParameter,(char *)err);       //显示错误信息
return 0;
}
```

用户运行上面的程序后，会在进度条上方出现当前播放文件的时间长度和已播放文件的时间长度，如图 9.22 所示。



图 9.22　显示文件的播放时间

在本节中，主要向用户讲述了实例中主要线程的分配以及各个线程的具体实现方法。通过本节的学习，用户将熟练使用 MCI 函数获取当前 MP3 文件的播放状态以及线程的运行等相关知识。

9.5.2　线程间通信

在 VC 中，实现线程间通信的方法有两种，分别是使用全局变量和使用线程消息。在

本实例中，将选择使用全局变量实现线程间的通信。用户在实例程序中，可以根据播放列表中的音乐序列号判断 MP3 的播放顺序。首先，用户需要在程序中定义一个整型变量 index。代码如下：

```
int index=0;                                    //定义并初始化全局变量
CMyDlg::CMyDlg(CWnd* pParent /*=NULL*/)
: CDialog(CMyDlg::IDD, pParent)
{
...                                             //省略部分代码
   }
```

然后，在列表控件的双击消息处理函数 OnDblclkList2()中，用户需要将列表选择项的索引赋值给全局变量 index。代码如下：

```
void CMyDlg::OnDblclkList2(NMHDR* pNMHDR, LRESULT* pResult)
{
...                                                     //省略部分代码
POSITION pos=m list.GetFirstSelectedItemPosition();
                                                        //获取用户当前选中的项目位置
if(pos==NULL)                                           //判断列表中是否为空
{
    MessageBox("列表为空！");                           //提示用户列表为空
}"
else
{
    int nItem=m_list.GetNextSelectedItem(pos);//获取用户选择的项索引
    index=nItem;                                        //将索引值赋予全局变量
...                                                     //省略部分代码
}}
```

在代码中，用户使用了列表控件类的成员函数 GetFirstSelectedItemPosition()和 GetNextSelectedItem()获取当前双击位置的项目索引值。这两个函数的原型如下：

```
POSITION GetFirstSelectedItemPosition( ) const;
int GetNextSelectedItem( POSITION& pos ) const;
```

其中，函数 GetFirstSelectedItemPosition()的作用是获取用户当前选择列表项的位置，其返回类型为 POSITION。而函数 GetNextSelectedItem()则是根据该列表项的位置获取其索引值，其参数 pos 表示列表项的位置。

用户在列表中双击某项后，则该项索引值便被程序记录在全局变量 index 中了。当用户单击“上一首”或者“下一首”按钮时，程序就可以根据该全局变量判断播放曲目的位置与顺序。有了这个全局变量，程序中便会按照列表中的顺序或者用户所单击位置的曲目进行播放。

9.6　数据读取与播放控制

在实例中，读取数据是指程序在启动时，应该从指定文件中读取默认的歌曲列表到列表控件中进行显示。而播放控制则是为播放器界面中各个功能按钮的消息响应函数添加相

应功能的代码。在本节中，主要向用户介绍播放器主要功能的代码编写及其数据的存储方法。

9.6.1　读取数据

如果播放器程序是第一次启动，则用户应该在指定目录中创建相应文件对列表中的曲目相关数据进行存储。其中，主要是存储曲目数据的名称、大小、类型以及文件路径。

1．封装曲目数据结构

在本实例中，用户为了方便对曲目的相关数据进行操作，应该将这些数据封装在一个结构体中。这样，用户使用起来比较方便。首先，将该结构体命名为 mp3data，结构定义如下：

```
typedef struct mp3                    //定义 MP3 文件数据结构体
{
  char heade[3];                      //TAG 字符标记
char title[30];                       //音乐文件名称
char arti [30];                       //演唱者
CString str;                          //路径字符串
}mp3data;
```

在该结构体中，主要的成员变量是 str，其表示 MP3 文件的完整路径名。程序播放 MP3 音乐时均根据该路径搜索相应的 MP3 文件。

然后，用户在主对话框类中定义该自定义结构体的变量。代码如下：

```
class CMyDlg : public CDialog
{
public:
    mp3data mp3d;                     //定义结构体变量
     ...                              //省略部分类变量
 }
```

用户定义结构体变量 mp3d 成功以后，便可以在该实例程序的任何地方使用该变量。

2．操作歌曲列表文件

用户在实例程序中，需要在程序启动时创建文件对象并且从该文件中读取相应数据。如果文件对象为空，表示程序是第一次启动，则用户需要创建文件。否则，程序读取该文件组中的数据并将这些数据显示到列表控件中。代码如下：

```
BOOL CMyDlg::OnInitDialog()
{
    CDialog::OnInitDialog();
CFile file1("歌曲列表.lw",CFile::modeReadWrite|CFile::modeCreate);
                                      //创建文件
if(file1.m hFile==NULL)               //若文件对象为空
{
    CFile file1("歌曲列表.lw",CFile::modeReadWrite|CFile::modeNoTruncate);
                                      //创建文件但不覆盖原有文件
```

```
}
else                                   //若文件已经存在
{
...                                    //省略部分代码
  }
```

在代码中，用户首先创建文件对象并且将该对象以可读可写方式关联到指定文件。如果该文件对象句柄为空，则表示文件并不存在，用户需要新建该文件。否则，用户便可以对该文件进行读取操作。代码如下：

```
while(mpd.title!=NULL)                              //判断获取的文件标题
{
    file1.Read(&mpd,sizeof(mpd));                   //读取文件数据
int nRow = m_list.InsertItem(1,mpd.title);          //在列表中插入行
   m_list.SetItemText(nRow,1,mpd.arti);             //设置数据
    if(mpd.heade && "TAG")                          //表示该文件为MP3 文件
{
   CString str="MP3";                               //添加字符串
m_list.SetItemText(nRow,2, str);                    //设置数据
}
}
file1.Close();                                      //关闭文件对象
```

用户使用以上代码对“歌曲列表”文件进行循环读取，直到该文件结束为止，并且用户将每次读取到的文件数据显示在列表控件中，如图 9.23 所示。



图 9.23　显示“歌曲列表”文件数据

通过本节的学习，用户可以熟练地使用列表文件进行数据的读取与存储操作。自定义结构体 mp3data 是本实例中非常重要的一个数据结构，用户必须学会熟练使用并构造其中的成员变量。

9.6.2　保存数据

当用户使用播放器界面中“添加歌曲”按钮向播放器中添加歌曲时，程序应该将该歌曲文件的相应信息显示在列表控件中和“歌曲列表”文件中进行存储。首先，在添加歌曲按钮的消息响应函数中添加打开文件对话框的功能。代码如下：

```
void CMyDlg::OnAdd()
{
CString strpath="MP3 音乐(*.mp3)|*.mp3||";            //过滤文件列表
CFileDialog filed(true,NULL,NULL,OFN HIDEREADONLY,strpath,NULL);
                                                     //创建文件对象并打开
if(filed.DoModal()==IDOK)                            //显示文件打开对话框
{
...                                                  //省略部分代码
}}
```

在 VC 中运行上面的代码以后，用户在界面中单击“添加歌曲”按钮，便会弹出一个“打开”对话框，如图 9.24 所示。

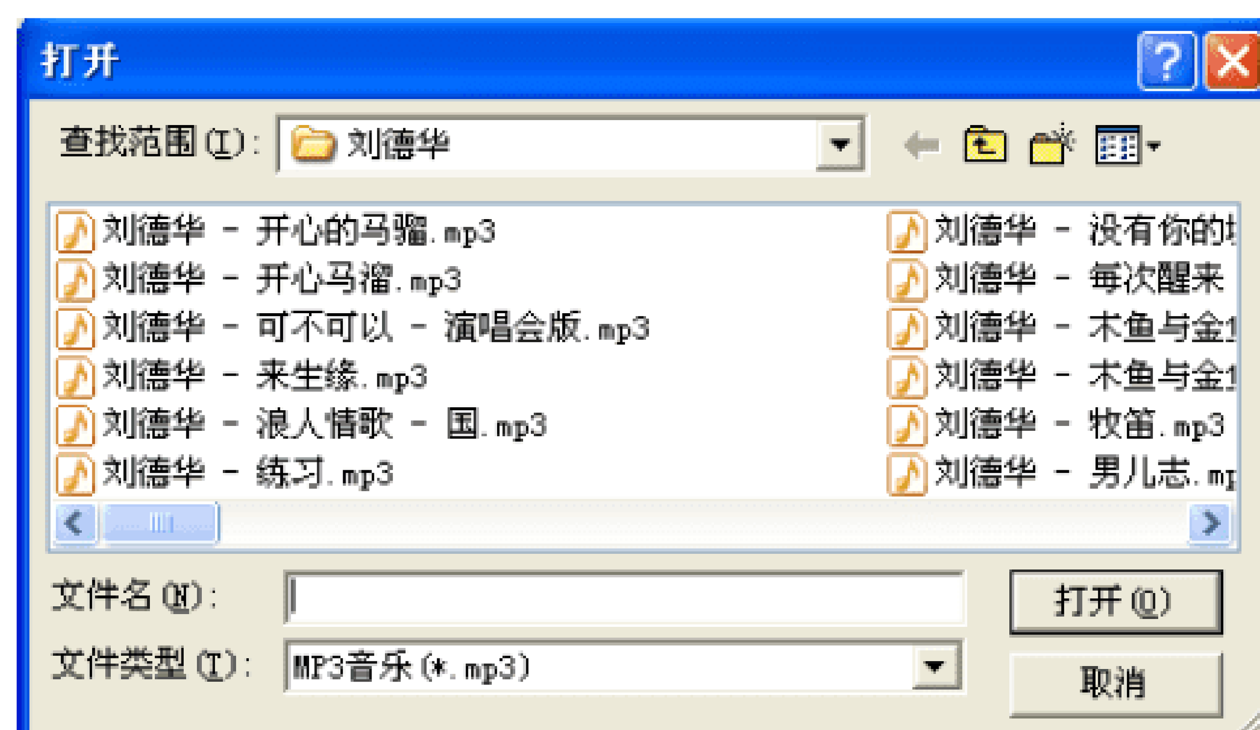


图 9.24　“打开”对话框

如果用户在“打开”对话框中选择需要添加的歌曲后，程序会在播放列表中添加相应的文件相关信息，并且程序会向“歌曲列表”文件中写入该歌曲文件的这些信息。这样程序在读取时，在列表控件中会显示这些默认的歌曲列表。

首先，用户需要创建两个文件对象并分别关联“歌曲列表”文件以及用户通过“文件打开”对话框所选择的文件。然后，用户从选择的 MP3 文件中读取相关信息到结构体变量 MP3 中，再将该变量中的全部内容写入“歌曲列表”文件中存储。代码如下：

```
...                                                //省略部分代码
CFile file1("歌曲列表.lw",CFile::modeReadWrite);//创建文件对象并关联该文件
POSITION pt=filed.GetStartPosition();              //获取用户选择的文件路径
CString path=filed.GetNextPathName(pt);
CFile file(path,CFile::modeReadWrite);             //创建文件对象并关联该文件
file.Seek(-128,CFile::end);                        //从文件结尾处移动文件指针
file.Read(&mp3,128);                               //读取文件内容
file.Close();                                      //关闭文件
strcpy(&mpd.title[0],&mp3.title[0]);               //复制各个结构体成员变量
strcpy(&mpd.arti[0],&mp3.arti[0]);
strcpy(&mpd.heade[0],&mp3.heade[0]);
if(mpd.heade && "TAG")                             //判断是否为 MP3 文件
```

```
    {
        CString str="MP3";
        strcpy(&mpd.heade[0],str.GetBuffer(1));
    }
    mpd.str=path;                               //获取 MP3 文件的路径
file1.Seek(2,CFile::end);
file1.Write(&mpd,sizeof(mp3));                  //将结构体数据写入列表文件中
file1.Flush();                                  //强制写入数据
}
file1.Close();                                  //关闭文件
...                                             //省略部分代码
```

通过以上代码，用户实现了将歌曲文件的标题以及该文件路径等信息写入“歌曲列表”文件中进行存储。待实例程序启动时会对该文件进行读取，并将这些歌曲信息显示到列表控件中。

最后，用户还必须将刚选择的 MP3 文件的相关信息显示在当前播放列表的最后，表示该 MP3 文件是用户最近添加到程序中的。代码如下：

```
...                                             //省略部分代码
int nRow = m_list.InsertItem(m_list.GetItemCount()+1,mp3.title);
                                                //插入行
m_list.SetItemText(nRow,1,mp3.arti);            //设置数据
if(mp3.heade && "TAG")                          //判断文件是否为 MP3 文件
{
    CString mp3="MP3";
m_list.SetItemText(nRow,2,mp3);                 //设置数据
}
```

在上面的程序中，用户将再一次判断所添加的文件是否为 MP3 文件。如果是，则在播放列表中的歌曲类型中显示字符串 MP3。否则，实例程序将不对该文件做任何处理，如图 9.25 所示。



图 9.25　将添加的 MP3 文件信息显示到播放列表中

用户保存 MP3 数据主要是遵循自定义结构体 mp3data 的成员定义顺序进行存储的，因为程序在启动的时候是按照该结构体进行数据读取操作。当用户保存数据时，如果没有遵循这个规律，则程序读取数据将发生错误。

9.6.3 识别数据文件信息

识别数据文件信息主要是指用户对 MP3 数据格式的识别。在本章 9.1 节中向用户介绍了 MP3 文件的标签帧位于该文件的最后 128 字节，并且列出了其中主要的数据成员。其中，MP3 标签帧是以字符串 TAG 开头。然后，该文件的其他信息都是以一定的顺序进行排列，以便用户进行读取与保存。其定义顺序的代码如下：

```
typdef struct mp3_struct                    //自定义 MP3 结构体
{
char heade[3];                              //TAG 字符标记
char title[30];                             //音乐文件名称
char arti [30];                             //演唱者
char alb [30];                              //专辑
char year[4];                               //出版年份
char text[28];                              //备注内容
char reser;                                 //保留
char tra;                                   //音轨
char genr;                                  //文件类型
 } mp3struct;
```

用户在实际编程时，便可以按照这个结构所定义的顺序对 MP3 数据进行读取和保存。其中，最后 3 个成员变量在 9.1 节中并未介绍。原因是这 3 个成员变量在实际编程时很少使用，一般都设置为 0。例如，当用户需要判断所操作的文件是不是 MP3 文件时，只需要判断该文件的标签帧中的第一个成员变量 heade 即可。如果该成员是字符串“TAG”，则该文件是 MP3 文件。否则，该文件不是 MP3 文件。

9.6.4 播放控制

在本节中，将编写详细而完整的代码向用户讲解各个功能控件的具体实现方法。例如，播放、停止以及暂停等功能。

1．实现播放功能

用户需要通过播放按钮播放音乐。所以，用户应该获取当前列表中被选择的 MP3 文件的路径，并且根据该文件路径调用 MCI 函数进行播放。代码如下：

```
void CMyDlg::OnPlay()                                //播放按钮消息响应函数
{
MCI_OPEN_PARMS open={0};                             //定义并初始化结构体
char str1[100];                                      //定义字符数组
POSITION pos=m_list.GetFirstSelectedItemPosition();//获取用户选择的位置
if(pos==NULL)                                        //如果选择为空
{
    MessageBox("当前没有选择！");
```

```
}
else                                                    //如果选择不为空
{
        int nItem=m_list.GetNextSelectedItem(pos);//获取列表中当前的选择项
CString str=m_list.GetItemText(nItem,3);        //获取当前选择项的文字
open.lpstrElementName=str;                      //指定播放文件路径
open.lpstrDeviceType="mpegvideo";               //指定播放设备
DWORD err;                                      //定义错误信息
err=mciSendCommand(0,MCI_OPEN,MCI_OPEN_TYPE|MCI_OPEN_ELEMENT|MCI_WAIT,(
DWORD)(LPVOID)&open);                           //初始化音频设备
if(err==0)                                      //如果初始化设备成功
{
MCI_PLAY_PARMS play;                            //定义结构体变量
play.dwFrom=0;                                  //指定播放位置为起始位置
play.dwCallback=NULL;                           //返回消息的窗口句柄为 NULL
mciSendCommand(open.wDeviceID,MCI_PLAY,0,(DWORD)&play);
                                                //播放指定文件
}
else                                            //初始化失败
{
    mciGetErrorString(err,(LPSTR)str1,100);     //获取失败信息
    MessageBox(str1);                           //显示失败信息
}
}
```

在代码中，用户首先通过调用列表控件的相关成员函数获取当前列表中被选择列表项，并从该项中获取指定列的文件路径信息。然后，将该文件路径字符串赋值给结构体 MCI_OPEN_PARMS 变量的相关成员即可。最后，调用 MCI 函数对该 MP3 文件进行播放。

2. 添加歌曲

用户可以通过界面中的添加歌曲按钮，将相应的 MP3 文件添加到列表中和列表文件中进行保存。这样，当程序启动时，可以将用户已经添加的音乐列表全部显示出来，还可以将用户刚选择的 MP3 文件信息显示到列表中。代码如下：

```
CString strpath="MP3 音乐(*.mp3)|*.mp3||";       //过滤文件列表
CFileDialog filed(true,NULL,NULL,OFN_HIDEREADONLY,strpath,NULL);
                                                //创建文件对象并打开
if(filed.DoModal()==IDOK)                       //显示文件打开对话框
{
CFile file1("歌曲列表.lw",CFile::modeReadWrite);//创建文件对象并关联该文件
POSITION pt=filed.GetStartPosition();           //获取用户选择的文件路径
CString path=filed.GetNextPathName(pt);
CFile file(path,CFile::modeReadWrite);          //创建文件对象并关联该文件
file.Seek(-128,CFile::end);                     //从文件结尾处移动文件指针
file.Read(&mp3,128);                            //读取文件内容
file.Close();                                   //关闭文件
strcpy(&mpd.title[0],&mp3.title[0]);            //复制各个结构体成员变量
strcpy(&mpd.arti[0],&mp3.arti[0]);
strcpy(&mpd.heade[0],&mp3.heade[0]);
if(mpd.heade && "TAG")                          //判断是否为 MP3 文件
    {
        CString str="MP3";
        strcpy(&mpd.heade[0],str.GetBuffer(1));
```

```
    }
    mpd.str=path;                                  //获取 MP3 文件的路径
file1.Seek(2,CFile::end);
file1.Write(&mpd,sizeof(mp3));                     //将结构体数据写入列表文件中
file1.Flush();                                     //强制写入数据
}
file1.Close();                                     //关闭文件
int nRow = m_list.InsertItem(m_list.GetItemCount()+1,mp3.title);//插入行
m_list.SetItemText(nRow,1,mp3.arti);               //设置数据
if(mp3.heade && "TAG")                             //判断文件是否为 MP3 文件
{
    CString mp3="MP3";
m_list.SetItemText(nRow,2,mp3);                    //设置数据
}
```

该按钮的功能是将用户所选择的 MP3 文件保存在列表和列表文件中。首先，用户单击“添加歌曲”按钮后，程序会弹出一个“打开”对话框。该对话框根据用户的选择会返回相应的文件路径等相关信息。

然后，用户可以根据所选择的文件路径调用 CFile 类的相关函数创建相应的文件对象并对其进行读取操作。如果用户操作成功，则在列表和列表文件中写入读取到的 MP3 数据即可。

3．更换播放歌曲

如果用户需要更换所播放的歌曲，可以通过单击“上一首”、“下一首”按钮或者双击播放列表中需要播放的歌曲，即可实现该功能。在本节中，将向用户介绍这几个功能的具体实现方法。

首先，用户可以通过单击“上一首”和“下一首”按钮更换当前播放的歌曲。用户在实际编程中，这一功能均是根据前面小节中所定义的全局变量 index 的改变而实现的。代码如下：

```
void CMyDlg::OnPre()                            //上一首按钮消息响应函数
{
CString str;                                    //定义字符串变量，用于获取播放路径
str=m_list.GetItemText(index-1,3);              //获取播放列表中当前选项的前一项
if(open.wDeviceID)                              //判断 MCI 设备 ID 是否存在
{
  mciSendCommand(open.wDeviceID,MCI_CLOSE,0,0);//若存在，则调用函数关闭
}
open.lpstrElementName=str;                  //若不存在，则赋予用户获取的播放路径
open.lpstrDeviceType="mpegvideo";           //指定 MCI 设备类型
err=mciSendCommand(0,MCI_OPEN,MCI_OPEN_TYPE|MCI_OPEN_ELEMENT|MCI_WAIT,(
DWORD)(LPVOID)&open);                       //打开并初始化设备
if(err==0)                                  //若没有发生设备初始化失败
{
MCI_PLAY_PARMS play;                        //定义播放结构体变量
mciSendCommand(open.wDeviceID,MCI_PLAY,0,(DWORD)&play);
                                            //调用函数播放指定的 MP3 文件
}
}

void CMyDlg::OnNext()                       //下一首按钮消息响应函数
```

```
{
    CString str;                    //定义字符串变量
str=m_list.GetItemText(index+1,3);//获取当前播放列表的下一项的播放路径
if(open.wDeviceID)                  //判断 MCI 设备 ID 是否存在
{
  mciSendCommand(open.wDeviceID,MCI_CLOSE,0,0);//若存在，则关闭指定 MCI 设备
}
open.lpstrElementName=str;          //若不存在，则赋予新获取的播放路径
open.lpstrDeviceType="mpegvideo";   //指定 MCI 播放设备 ID
err=mciSendCommand(0,MCI_OPEN,MCI_OPEN_TYPE|MCI_OPEN_ELEMENT|MCI_WAIT,(
DWORD)(LPVOID)&open);               //调用函数打开并初始化 MCI 设备
if(err==0)                          //若没有发生错误，则播放指定 MP3 文件
{
MCI_PLAY_PARMS play;                //定义 MCI 播放结构体变量
mciSendCommand(open.wDeviceID,MCI_PLAY,0,(DWORD)&play);
                                    //调用函数打开指定的 MCI 设备播放 MP3
}
}
```

在代码中，分别实现了上一首和下一首按钮的消息响应函数。其中，在上一首按钮的消息响应函数中是将全局变量 index 减去 1。以便用户获取播放列表中对应项的播放文件路径，再以该文件路径为参数调用相关的 MCI 函数播放该 MP3 文件即可。而在下一首按钮的消息响应函数中是将全局变量 index 加 1，其他实现方法均与上一首按钮的消息响应函数实现方法一样。

然后，用户也可以通过在播放列表中双击对应的 MP3 文件名，实现更换播放歌曲的功能。代码如下：

```
void CMyDlg::OnDblclkList2(NMHDR* pNMHDR, LRESULT* pResult)
{
CString str;                                    //定义字符串变量
POSITION pos=m_list.GetFirstSelectedItemPosition();
                                                //获取用户当前选择的列表项位置
if(pos==NULL)                                   //判断选择的位置是否为空
{
    MessageBox("用户双击的位置错误或该列表为空！（用户双击位置应该为每行第一列）");
}
else                                            //若不为空
{
int nItem=m_list.GetNextSelectedItem(pos);      //获取用户选择的列表项索引
index=nItem;                                    //将该索引值赋给全局变量
index
str=m_list.GetItemText(nItem,3);                //获取指定列表项的MP3文件路径
if(open.wDeviceID)                              //判断 MCI 设备 ID 是否存在
{
  mciSendCommand(open.wDeviceID,MCI_CLOSE,0,0);//若存在，则关闭该 MCI 设备
}
open.lpstrElementName=str;                      //保存获取到的 MP3 文件路径
open.lpstrDeviceType="mpegvideo";               //指定 MP3 播放设备类型
err=mciSendCommand(0,MCI_OPEN,MCI_OPEN_TYPE|MCI_OPEN_ELEMENT|MCI_WAIT,(
DWORD)(LPVOID)&open);                           //打开并初始化 MCI 设备
if(err==0)                                      //若没有发生任何错误
{
MCI_PLAY_PARMS play;                            //定义 MCI 播放结构体变量
mciSendCommand(open.wDeviceID,MCI_PLAY,0,(DWORD)&play);
                                                //播放指定 MP3 文件
```

```
    }
  }
}
```

4．实现暂停播放

与其他播放器一样，本实例中所示的播放器程序也具有暂停播放和继续播放功能。本节中，将这两个功能放到同一个函数中实现，即暂停按钮的消息响应函数 OnZanting()中进行实现。

在实例程序中，暂停播放和恢复播放功能的实现原理是当用户单击该按钮时，会将变量 n 赋值为 1 表示用户暂停该文件的播放，并且调用函数 mciSendCommand()实现暂停功能。而当用户再次单击该按钮时，程序会将年龄 n 赋值为 0 表示用户继续播放，并且调用 MCI 函数恢复被暂停的播放功能。代码如下：

```
void CMyDlg::OnZanting()
{
    CString str;
GetDlgItem(IDC_ZANTING)->GetWindowText(str);
if(n==0)
    {
        mciSendCommand(open.wDeviceID,MCI_PAUSE,0,0);
                                                    //向设备发送暂停命令
        GetDlgItem(IDC_ZANTING)->SetWindowText("继续");
        n=1;
    }
else
    {
        mciSendCommand(open.wDeviceID,MCI_RESUME,0,0);
                                                    //向设备发送继续播放命令
        GetDlgItem(IDC_ZANTING)->SetWindowText("暂停");
        n=0;
    }
}
```

用户将上面的程序编译、运行之后可以看到这些代码的实际运行状态。程序运行时，用户单击“暂停”按钮，如图 9.26 所示。



图 9.26　用户暂停播放功能

当用户单击“暂停”按钮时，会看到该按钮上的文字会被修改为继续，同时正在播放中的 MP3 歌曲也会被暂停播放。当用户再次单击该按钮时，被暂停播放的 MP3 文件又会重新播放，并且程序会将该按钮上的文字修改为暂停，如图 9.27 所示。



图 9.27　用户继续播放歌曲

在实例程序中，用户可以通过单击“暂停”按钮实现 MP3 文件的播放状态与暂停状态的相互切换功能。但是，这种切换仅仅局限于同一个 MP3 文件。

5．实现停止播放功能

在实例程序中，用户还应该为其添加停止功能，用于停止播放当前所播放的歌曲。首先，用户需要为实例界面中的“停止”按钮添加相应的消息响应函数 OnStop()。然后，在该函数中调用函数 mciSendCommand()向当前 MCI 设备设置停止指令标志 MCI_STOP。代码如下：

```
void CMyDlg::OnStop()
{
    mciSendCommand(open.wDeviceID,MCI_STOP,0,0);//向当前 MCI 设备设置指令标志
  index+=1;                                    //当前列表项加 1
  }
```

在程序中，用户首先调用函数 mciSendCommand()向当前 MCI 设备发送了指令标志 MCI_STOP，表示停止当前所播放的 MP3 文件。用户运行程序，结果如图 9.28 所示。

当用户单击“停止”按钮以后，实例程序会停止当前所播放的曲目，并且将播放列表的索引加 1 指向下一个曲目索引。这样，当用户重新播放曲目时，实例程序所播放的应该是用户所停止的曲目的下一个曲目。例如，在图 9.28 中，当用户停止播放后，再次单击“播放”按钮时，实例程序会播放下一个曲目，也就是列表中的第二个“心蓝”。



图 9.28　用户停止当前播放曲目

在实例程序中，所有的播放控制功能已经基本实现。请用户结合随书光盘中相应章节的代码与运行实例进行学习。在 9.7 节中，将向用户介绍通过搜索 MP3 文件的方法向播放列表中添加曲目。

9.7　实现搜索功能

一般情况下，为了方便用户的使用，播放器都应该具有搜索相应文件的功能。因此，在该实例中，也需要为用户添加搜索功能。本实例中的搜索功能分为“本目录搜索”和“本地搜索”两种。其中，前者主要是在播放器所在的目录下搜索 MP3 文件，而后者则是在所有盘符上搜索 MP3 文件。如果搜索成功，则程序会将这些 MP3 文件添加到播放列表中，并且将其主要信息写入相应的文件中保存。

9.7.1　相关类和函数说明

在 VC 中，用户可以调用 MFC 类 CFileFind 的相关成员函数或者 API 函数 FindFirstFile()和 FindNextFile()实现文件搜索功能。

1．CFileFind类实现搜索功能

CFileFind 类是 MFC 中封装的关于文件搜索的相关类。其构造函数原型如下：

```
CFileFind();
```

该函数的作用是创建一个 CFileFind 类的实例对象。例如，用户在程序中创建 CFileFind 类对象，代码如下：

```
...                                                   //省略部分代码
CFileFind findfile;                                   //创建文件搜索对象
```

如果文件搜索对象创建成功，则用户可以调用其成员函数实现相应的功能。其中，函数 FindFile()和 FindNextFile()是该类中非常重要的两个函数。原型分别如下：

```
virtual BOOL FindFile( LPCTSTR pstrName = NULL, DWORD dwUnused =0);
virtual BOOL FindNextFile();
```

如果函数 FindFile()执行成功，则返回非 0 值。否则，函数将返回 0，表示执行失败。其参数 pstrName 表示需要搜索的文件名，如果该参数为 NULL，则表示搜索所有文件。参数 dwUnused 必须设置为 0。例如，用户使用这个函数搜索所有的 MP3 文件，代码如下：

```
...                                                   //省略部分代码
CFileFind findfile;                                   //创建文件搜索对象
findfile. FindFile("*.mp3",0);                        //搜索 MP3 文件
...                                                   //省略部分代码
```

函数 FindNextFile()表示搜索下一个文件。该函数必须与 FindFile()一起使用。否则，用户将不能实现文件搜索功能。代码如下：

```
...                                                   //省略部分代码
CFileFind findfile;
CString str;                                          //定义字符串对象
if(findfile.FindFile("F:\\音乐\\11\\*.*",0))          //搜索所有文件
{
  while(findfile.FindNextFile())                      //循环搜索文件
  {
     str+="文件路径：";                                //格式化字符串对象
str+=findfile.GetFilePath();                          //获取文件路径
 MessageBox(str);                                     //显示信息
 str=" ";
}}
...                                                   //省略部分代码
```

将上面的成功程序运行之后，用户单击“搜索目录歌曲”按钮后，程序会不停地显示搜索到的文件路径，如图 9.29 所示。

当用户搜索到相应的文件后，可以调用 CFilefind 类的其他函数获取文件的相关信息。

注意： 由于本章中实现搜索功能主要是使用 API 函数，所以，对 CFilefind 类的讲解较为简略。

2. API函数实现搜索功能

用户使用 API 函数实现搜索功能时，需要使用到两个 API 函数，分别是 FindFirstFile()和 FindNextFile()。这两个函数的功能与 CFileFind 类中的成员函数 FindFile()和 FindNextFile()的功能相对应。函数原型如下：

```
HANDLE FindFirstFile(
  LPCTSTR lpFileName,
  LPWIN32 FIND DATA lpFindFileData
);
BOOL FindNextFile(
```

```
  HANDLE hFindFile,
  LPWIN32_FIND_DATA lpFindFileData
);
```



图 9.29　搜索文件并获取其路径

其中，函数 FindFirstFile()作用是搜索文件并返回搜索句柄。如果执行成功，则函数返回文件搜索的句柄。否则，函数将返回 NULL。其参数 lpFileName 表示搜索的文件名。参数 lpFindFileData 是指向结构体 WIN32_FIND_DATA 的指针。该结构体定义如下：

```
typedef struct _WIN32_FIND_DATA {
    DWORD dwFileAttributes;                          //文件属性
    FILETIME ftCreationTime;                         //文件创建时间
    FILETIME ftLastAccessTime;                       //文件最后访问时间
    FILETIME ftLastWriteTime;                        //文件最后保存时间
    ...                                              //省略部分不常用的成员变量
    TCHAR    cFileName[ MAX_PATH ];                  //文件路径
    TCHAR    cAlternateFileName[ 14 ];               //文件名
} WIN32_FIND_DATA;
```

函数 FindNextFile()的作用是继续搜索指定文件，并将文件的相关信息保存到结构体 WIN32_FIND_DATA 中。其参数 hFindFile 是函数 FindFirstFile()执行成功后所返回的文件搜索句柄。

例如，用户使用这两个 API 函数进行搜索 MP3 文件。代码如下：

```
...                                                  //省略部分代码
WIN32_FIND_DATA data;                                //定义结构体变量
BOOL boo;                                            //定义布尔变量
HANDLE file=:: FindFirstFile("F:\*.mp3",&data);//返回文件搜索句柄
if(file!=NULL)
{
 do{
CString str="文件路径：";                            //定义字符串对象
   str+=(CString)data. cFileName;                    //获取文件路径
```

```
int nRow=m_list.InsertItem(m_list.GetItemCount()+1,mp.mp3.title);//插入行
m_list.SetItemText(nRow,1,mp.mp3.arti);                  //设置数据
if(mp.mp3.heade && "TAG")
{
    CString mp3="MP3";
m_list.SetItemText(nRow,2,mp3);                          //设置数据
}
m list.SetItemText(nRow,3,str);
boo=:: FindNextFile(file,&data);                         //查找下一个文件
}while(boo);
}
```

上面的程序搜索到指定文件后，便将该文件添加到播放列表中存储，如图 9.30 所示。



图 9.30　添加搜索到的文件到播放列表中

当用户搜索完所有文件以后，需要调用函数 FindClose()关闭文件搜索句柄。函数原型如下：

```
BOOL FindClose(
  HANDLE hFindFile
);
```

参数 hFindFile 表示关闭的文件搜索句柄。例如，用户将前面搜索文件的句柄 file 关闭，代码如下：

```
    ...                                          //省略部分代码
::FindClose(file);                               //关闭文件搜索句柄
```

在本节中，主要向用户介绍了文件搜索的基本实现方法以及实现该方法所要使用的 MFC 类或 API 函数。

9.7.2　搜索本目录文件

在本节中，主要向用户讲述使用 MFC 类或 API 函数，在播放器所在的目录下搜索 MP3

文件的具体编程方法。

1. 使用MFC类搜索目录文件

下面用户使用 MFC 类 CFileFind 搜索播放器所在目录中的所有 MP3 文件。具体代码如下：

```
...                                                     //省略部分代码
CFileFind findfile;                                     //创建文件搜索对象
    CString str2;
findfile.FindFile("F:\\音乐\\11\\*.mp3",0); //搜索目录中的 MP3 文件
while(findfile.FindNextFile())
{
str2=findfile.GetFilePath();                            //获取文件路径
CFile file(str2,CFile::modeRead);                       //创建文件对象
file.Seek(-128,CFile::end);                             //从文件结尾移动指针
file.Read(&mp.mp3,128);                                 //读取文件
file.Close();                                           //关闭文件
int nRow=m_list.InsertItem(m_list.GetItemCount()+1,mp.mp3.title);
                                                        //在列表中插入行
m_list.SetItemText(nRow,1,mp.mp3.arti);                 //设置数据
if(mp.mp3.heade && "TAG")
{
    CString mp3="MP3";
m_list.SetItemText(nRow,2,mp3);                         //设置数据
}
m_list.SetItemText(nRow,3,str2);
mp.str[0]=str2.GetBuffer(0);
mp.mp3.text[28]=0;                                      //赋初值
mp.mp3.year[4]=0;
mp.mp3.alb[30]=0;
CFile file1("歌曲列表.lw",CFile::modeWrite|CFile::modeNoTruncate|CFile::
modeCreate);
                                                        //关联列表文件
file1.Seek(0,CFile::end);                               //定位文件指针到结尾
file1.Write(&mp,sizeof(mp));                            //写入文件
file1.Flush();                                          //强制写入文件
file1.Close();                                          //关闭文件
}
```

在程序中，用户首先定义了一个文件搜索对象实例 findfile。接着使用该对象调用其成员函数 FindFile()对路径“F:\\音乐\\11*.mp3”下的所有 MP3 文件进行搜索，并将搜索的结构添加到播放列表中。

在 VC 中编译并运行上面的程序，用户可以单击“搜索目录歌曲”按钮实现在本目录下搜索所有的 MP3 文件，如图 9.30 所示。

2. 使用API函数搜索目录文件

用户在实例程序中，除了可以使用 MFC 类进行 MP3 文件的搜索以外，还可以使用 API 函数进行。具体代码如下：

```
...                                                     //省略部分代码
    CFileFind findfile;                                 //定义文件查找对象
```

```
CString str2,strTmp;                              //定义字符串变量
CString strpath="";                               //定义路径字符串
BROWSEINFO bBinfo;                                //定义变量
    memset(&bBinfo,0,sizeof(BROWSEINFO));         //定义结构并初始化
    bBinfo.hwndOwner=m_hWnd;                      //设置对话框所有者句柄
     bBinfo.lpszTitle="请选择安装路径：";           //修改对话框标题
     bBinfo.ulFlags=BIF_RETURNONLYFSDIRS;         //设置标志只允许选择目录
     LPITEMIDLIST lpDlist;                        //定义目录列表变量
    lpDlist=SHBrowseForFolder(&bBinfo);           //显示选择对话框
    if(lpDlist!=NULL)
    {
      SHGetPathFromIDList(lpDlist,strTmp.GetBuffer(0));
                                                  //把项目标识列表转化成目录
strcat(strTmp.GetBuffer(0),"\\*.mp3");            //连接字符串
findfile.FindFile(strTmp);
if(file!=NULL)
{
 do{
CString str;
   str=data.cFileName;                            //获取文件路径
CFile file2(str,CFile::modeRead);                 //创建文件对象
file2.Seek(-128,CFile::end);                      //定位文件
file2.Read(&mp.mp3,128);                          //读取文件
file2.Close();                                    //关闭文件
int nRow=m_list.InsertItem(m_list.GetItemCount()+1,mp.mp3.title);//插入行
m_list.SetItemText(nRow,1,mp.mp3.arti);           //设置数据
if(mp.mp3.heade && "TAG")
{
    CString mp3="MP3";
m_list.SetItemText(nRow,2,mp3);                   //设置数据
}
m list.SetItemText(nRow,3,str);
mp.mp3.text[28]=0;                                //填充数据结构
mp.mp3.year[4]=0;
mp.mp3.alb[30]=0;
CFile file1("歌曲列表.lw",CFile::modeWrite|CFile::modeNoTruncate|CFile::
modeCreate);
                                                  //关联文件
file1.Seek(0,CFile::end);                         //定位文件指针
file1.Write(&mp,sizeof(mp));                      //写入文件
file1.Flush();                                    //强制写入
file1.Close();                                    //关闭文件
boo=:: FindNextFile(file,&data);                  //查找下一个文件
}while(boo);
::FindClose(file);                                //关闭文件搜索句柄
}
```

用户使用 API 函数搜索文件的原理与使用 MFC 类是一样的。上面的代码运行后会弹出“浏览文件夹”对话框，如图 9.31 所示。

当用户选择一个目录以后，程序会在该目录中查找所有的 MP3 文件，并将这些文件添加到播放列表中和写入文件中进行保存。

在本节中，分别向用户介绍了使用 MFC 类和 API 函数实现搜索所有 MP3 文件的方法和具体的代码实现，请用户将本节所讲内容结合随书光盘中的实例代码与运行结果一起进行学习。

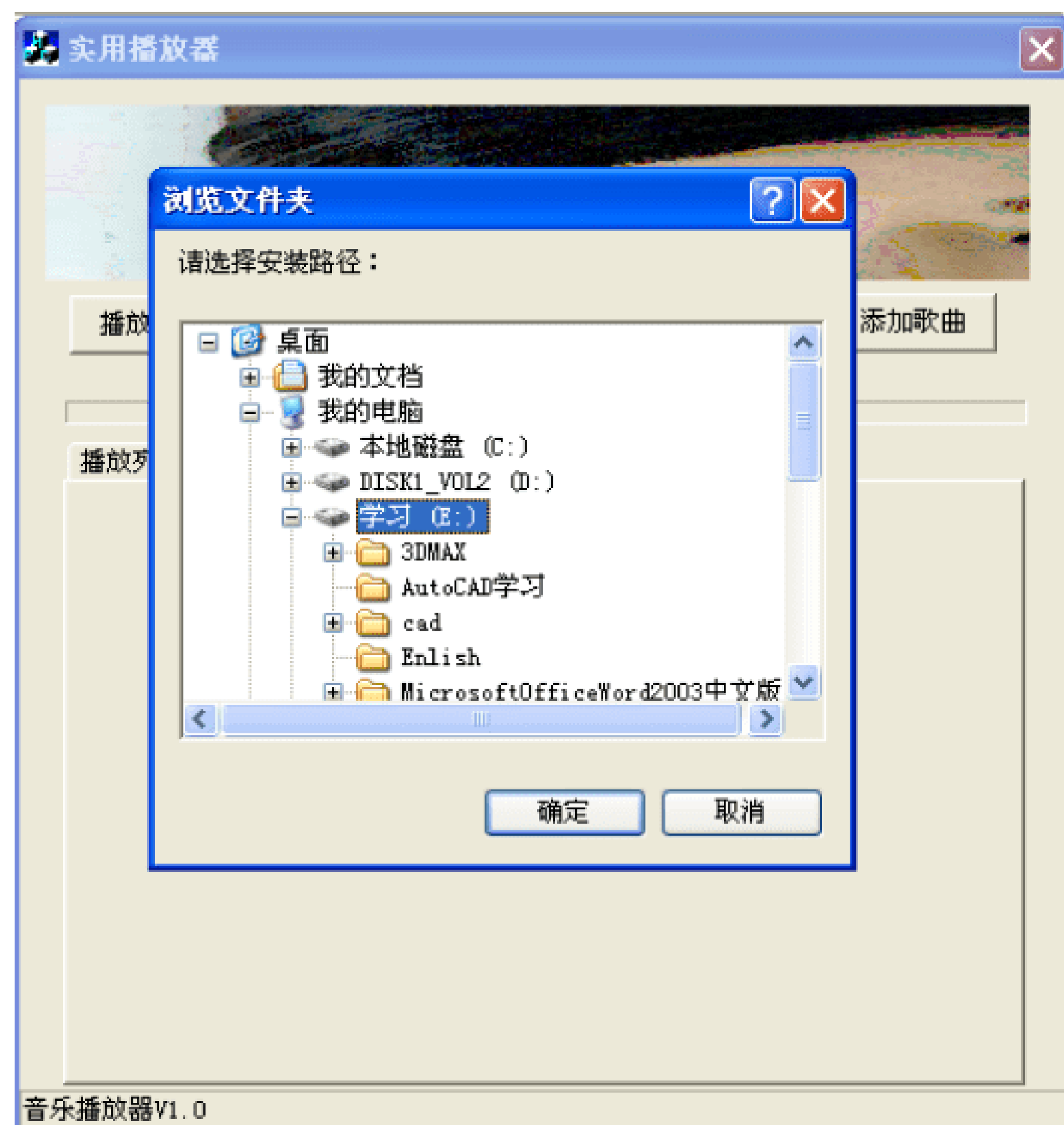


图 9.31 “浏览文件夹”对话框

9.7.3　搜索本地文件

在 9.7.2 节中，用户主要是实现在指定目录中搜索 MP3 文件。而在本节中，将向用户介绍在本地计算机中搜索所有 MP3 文件的方法。实现本地搜索所使用的方法主要是使用 MFC 中的 CFileFind 类。

首先，用户需要在程序中定义一个字符串数组，用于保存搜索路径。代码如下：

```
CString str[4]={"C:\\","D:\\","E:\\","F:\\"};  //指定搜索路径
...                                             //省略部分代码
```

然后，在搜索本地歌曲按钮的消息响应函数 OnSousuomulu2()中，添加相应代码实现搜索指定路径中的所有 MP3 文件。其搜索原理与搜索指定目录下的文件是一样的，代码如下：

```
...                                             //省略部分代码
CFileFind findfile;                             //创建文件搜索对象
    CString str2;
CString str[4]={"C:\\","D:\\","E:\\","F:\\"};  //指定搜索路径
findfile.FindFile("F:\\音乐\\11\\*.mp3",0);     //搜索目录中的 MP3 文件
for(int i=0;i<4;i++)
{
str3=str[i];
str3+="音乐\\11\\*.mp3";
findfile.FindFile(str3,0);
while(findfile.FindNextFile())
{
str2=findfile.GetFilePath();                    //获取文件路径
CFile file(str2,CFile::modeRead);               //创建文件对象
```

```
file.Seek(-128,CFile::end);                    //从文件结尾移动指针
file.Read(&mp.mp3,128);                        //读取文件
file.Close();                                  //关闭文件
int nRow=m list.InsertItem(m list.GetItemCount()+1,mp.mp3.title);
                                               //在列表中插入行
m list.SetItemText(nRow,1,mp.mp3.arti);        //设置数据
if(mp.mp3.heade && "TAG")
{
    CString mp3="MP3";
m_list.SetItemText(nRow,2,mp3);                //设置数据
}
m list.SetItemText(nRow,3,str2);
mp.str[0]=str2.GetBuffer(0);
mp.mp3.text[28]=0;                             //赋初值
mp.mp3.year[4]=0;
mp.mp3.alb[30]=0;
CFile file1("歌曲列表.lw",CFile::modeWrite|CFile::modeNoTruncate|CFile::
modeCreate);
                                               //关联列表文件
file1.Seek(0,CFile::end);                      //定位文件指针到结尾
file1.Write(&mp,sizeof(mp));                   //写入文件
file1.Flush();                                 //强制写入文件
file1.Close();                                 //关闭文件
}
```

在代码中，用户主要使用了循环变换搜索路径的方法对文件进行搜索。其他功能实现与程序在指定路径下搜索的实现方法一样。因此，本节将不再对该方法进行赘述，具体功能代码请参考随书光盘中的对应程序。

9.8　小　　结

在本章中，主要向用户讲解从 MP3 文件的基本结构知识到实例工程的创建、程序中各个功能的具体实现方法以及 MCI 函数的用法等。用户通过本章的学习，不但能对 MP3 格式的文件进行读取和修改，还可以利用 MCI 函数实现 MP3 文件的播放功能。请用户在学习完本章后，一定要结合随书光盘中的实例程序进行学习和调试程序。

第 10 章　P2P 网络播放器

P2P（peer-to-peer）表示网络中的对等传输协议。一般情况下，用户在实际生活中所使用的即时聊天软件等均采用的是 P2P 传输模式传输数据。在本章中，将主要向用户讲解使用 P2P 协议从本地客户机传输 MP3 文件到另一个客户机中进行播放的实现过程以及方法。

10.1　P2P 网络应用

在网络中，用户所使用的许多软件都是基于 P2P 协议进行开发的。例如，用户经常使用的一些下载软件，这种下载软件在下载的同时也在上传文件。因此，当使用这类下载软件的用户越多时，其下载速度就越快。在本节中，将向用户介绍 P2P 的基本概念和 P2P 网络模型。

10.1.1　P2P 概述

P2P 协议在实际使用时，常被称为点对点传输协议。使用该传输协议传输数据的双方既是服务器，又是客户机。因此，使用 P2P 传输协议传输数据的速度很快。但是，由于使用 P2P 协议进行数据传输的程序会对用户的计算机不断地进行读取操作。所以，使用 P2P 对用户的计算机有着一定的副作用。

现在，P2P 技术主要被大量地运用在下载软件等共享软件中。因为使用 P2P 技术实现这些共享软件时，该软件便具有了很多优点。

- 提高了资源数据的利用率。例如：当用户 A 通过 P2P 软件共享网络中的资源时，而用户 B、用户 C 可以通过 P2P 软件从用户 A 的计算机上得到各自所需要的数据资源。所以，用户使用 P2P 软件时，所使用的用户越多，则其数据传输的速度也就越快。

注意：当用户在实际编程时，实现这样的功能所使用的 P2P 网络模型是网状模型，该模型将在 10.1.2 节中讲解。

- 使用 P2P 技术开发的共享软件均有共同的一个特点，即使用高共享软件的用户越多，数据共享的数量越多，网络也就越稳定。该优点是基于 C/S 模式通信的软件所不能比拟的。
- 由于 P2P 技术采用对等网络技术。所以，使用 P2P 进行通信的双方或多方均是直接传输资源数据，使其传输速度加快。与 C/S 模式的通信软件相比，P2P 软件减少

了数据的中转。不但降低了成本，还使通信双方缩短了数据传输所用的时间。

P2P 软件的这些优点已经被广泛地得到实际应用。在日常生活中，用户经常会使用到这些类型的 P2P 软件。例如，PPLive、PPS、风行等网络直播软件。

但是，这些 P2P 软件在方便用户的同时，也给用户带来了许多未知的安全威胁。例如，用户 A 可能在其共享资源中添加病毒等有害程序。当用户 B 通过 P2P 软件共享数据时，用户 A 便将带有病毒的数据传送到用户 B 的计算机中，导致计算机中毒。这个病毒数据会被 P2P 软件传输到其他计算机上，并且其传播速度会非常快，以致用户没有察觉就已经中了病毒。

P2P 软件还有一个更大的缺点在于其在获取其他计算机数据的同时，也在为其他计算机提供共享数据。所以，P2P 软件会不停地在计算机上进行读写操作，这样会对用户的计算机造成一定的损害。

10.1.2　P2P 网络模型

在 10.1.1 节中，已经向用户介绍了 P2P 软件的应用范围以及其在实际应用中的优缺点。而 P2P 的这些优缺点也取决于 P2P 的网络模型。目前，用户在实际应用中，会经常使用到的 P2P 网络模型主要为网状模型，本章实例程序也将主要使用该模型。所以，在本节中将主要向用户介绍这种网络模型的相关基础知识。

1．网状模型概述

网状模型在 P2P 网络中，是指该网络拥有一台主计算机，而其他安装有 P2P 软件的计算机则称为网络节点。通常，在 P2P 网络的主计算机中，都会保存有该网络中各个节点计算机的地址以及所拥有的文件信息等。

如果某个节点计算机需要某些文件，则该节点计算机会向网络中的主计算机发送相关的查询信息。当主计算机接收到该查询信息以后，则在节点计算机列表中查询包含所请求文件的节点计算机地址。然后，主计算机再将该地址返回给发送查询请求的节点计算机。最后，通过主计算机返回的节点地址，两台节点计算机会直接进行连接、传输文件。

注意： 在 P2P 网状模型中，主计算机相当于一台查询服务器，并不真正拥有任何资源文件，只提供在该网络中所有节点计算机的地址和相应的存储文件信息。因此，基于网状模型开发的 P2P 软件并不是真正意义上的点对点传输软件。

用户在实际生活中，会发现几乎所有基于 P2P 网状模型开发的软件都有一个共同点，就是使用该软件的用户越多，共享数据的速度也就越快。这是因为使用的用户越多，则表示所共享的数据也相应增多。此时，P2P 软件则会选择一个空闲并且拥有相应文件的节点计算机发出连接请求，等待连接成功后便开始传输文件。

P2P 网状模型的优点是使每个网络节点计算机根据当前的网络状况，自由选择目标计算机。这样，不仅节省了用户的带宽，还缩短了数据操作的时间等。但是，由于 P2P 网状模型的通信方式决定，在该网络模型中的所有节点计算机都可能会遭到病毒文件的袭击。因此，用户使用基于该类模型开发的 P2P 软件时，应该非常小心，不要随意下载文件。

2. 网状模型结构

P2P 网状模型结构与 C/S 网络模型结构一样，也拥有固定的 P2P 网状模型结构，如图 10.1 所示。

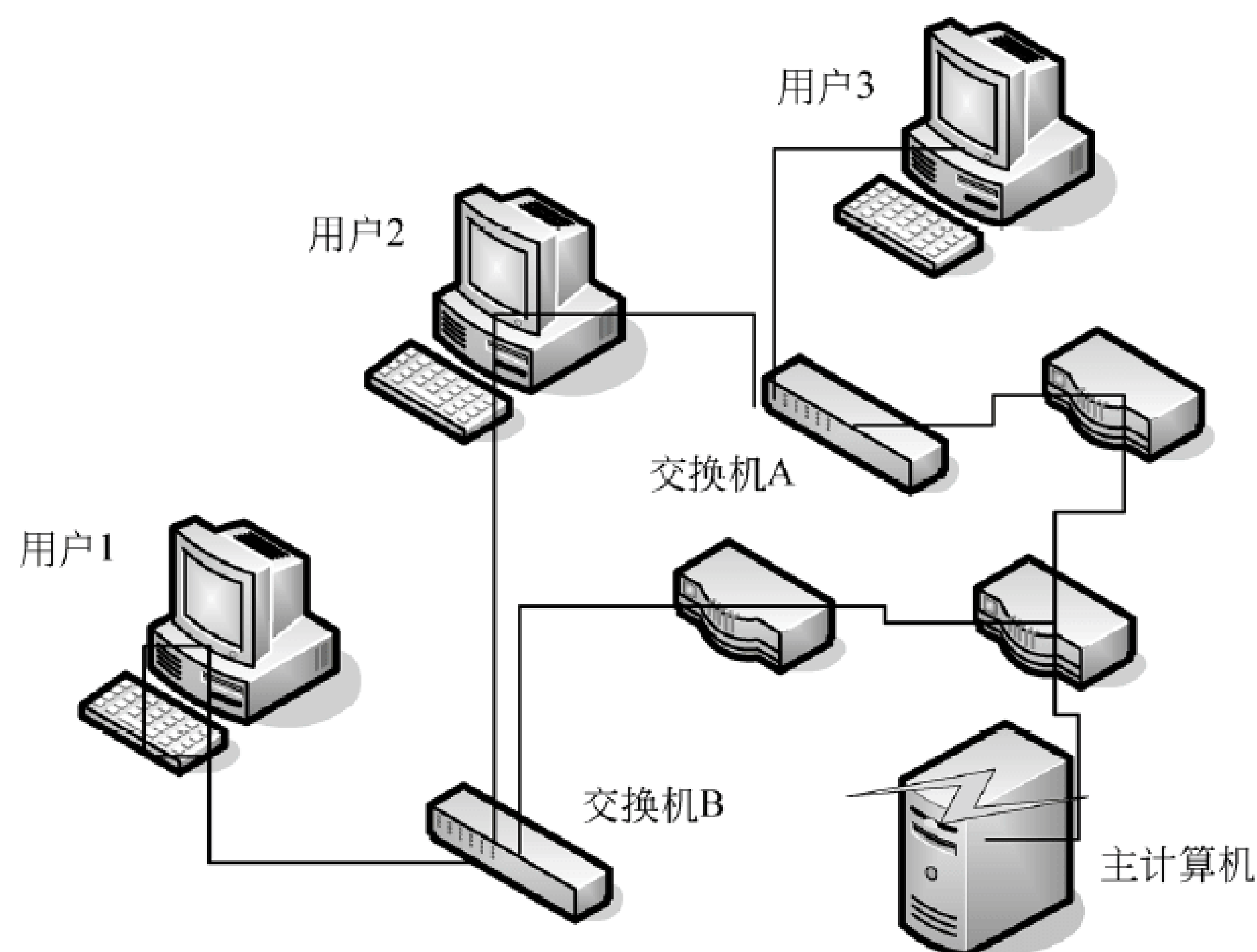


图 10.1　P2P 网状模型结构

在图 10.1 中，用户可以看到 3 个用户之间可以通过交换机 A 和交换机 B 互相进行数据传输。但是，3 个客户端必须在传输数据之前，向主计算机发送相关的信息，请求拥有相应文件存储的其他用户计算机地址。

由于 P2P 网状模型的优点，所以越来越多的网络视频直播软件均开始基于该模型设计。虽然，在实际生活中，各种 P2P 软件的开发模式多种多样。但是，均离不开 P2P 通信的基本方法。

在本节中，主要向用户介绍了 P2P 网状模型的基本通信方式、P2P 网状模型的优缺点以及网状模型的基本结构。用户在学习本节基础知识时，一定要体会其中的知识点，例如，P2P 网状模型的通信方式等。

10.2　界 面 设 计

本节将根据前面所介绍的关于 P2P 网状模型的通信方式，在 VC 中编写代码实现 P2P 网络中的各节点计算机相互寻址，相互连接以及相互传输数据等。

10.2.1　创建工程

由于本章所讲解的 P2P 网络播放器运行时，需要 Windows Sockets 的支持。所以，用户在创建实例工程时，必须选择工程支持 Windows Sockets 功能。否则，用户只能在代码

中手动添加该功能。工程创建的基本步骤如下。

（1）用户打开 VC，选择“文件”|“新建”命令，打开“新建”对话框，如图 10.2 所示。用户需要在工程类型列表中选择 MFC AppWizard[exe]选项，表示创建的工程项目基于 MFC 应用程序。在工程名称中修改该实例工程名为“P2P 网络播放器”，并选择工程文件的存储路径。

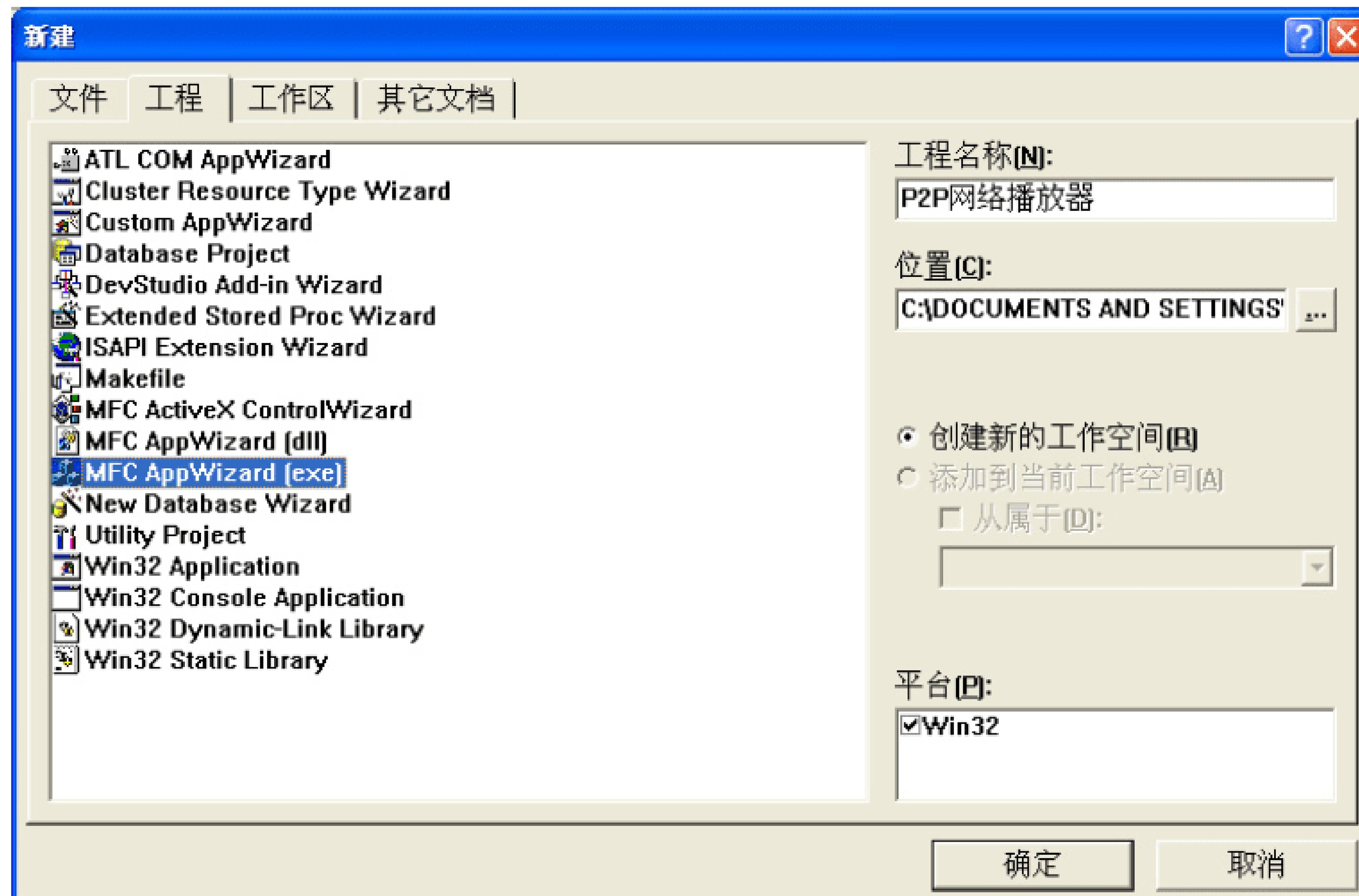


图 10.2　“新建”对话框

（2）单击“确定”按钮，进入工程设置第二步，选择应用程序的类型为“单文档”，如图 10.3 所示。

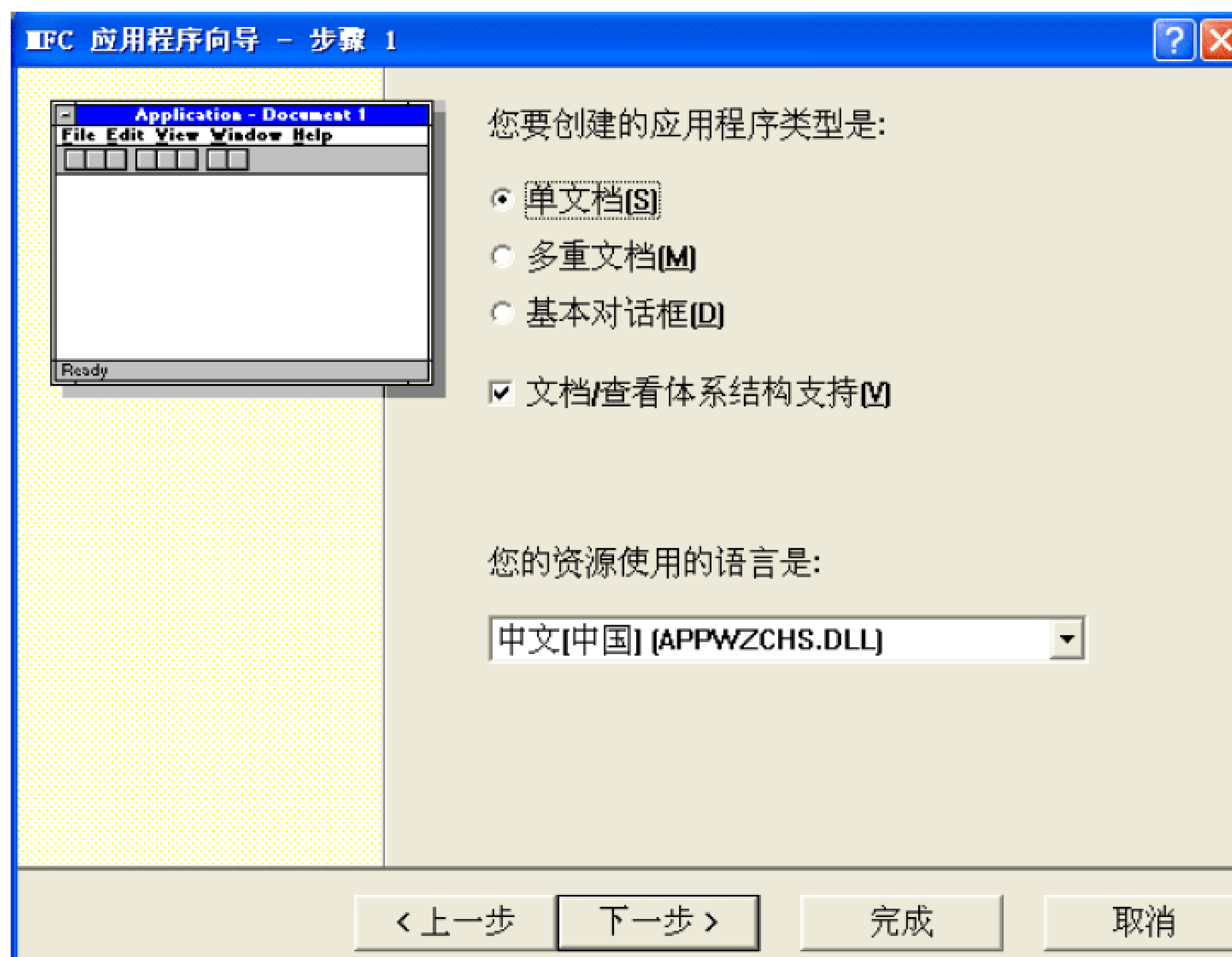


图 10.3　选择应用程序类型

注意：由于在该实例程序中，用户需要频繁地使用菜单。所以，用户在创建工程时，选择的应用程序类型为“单文档”。

（3）单击“下一步”按钮，选择该实例工程是否包含数据库，如图 10.4 所示。

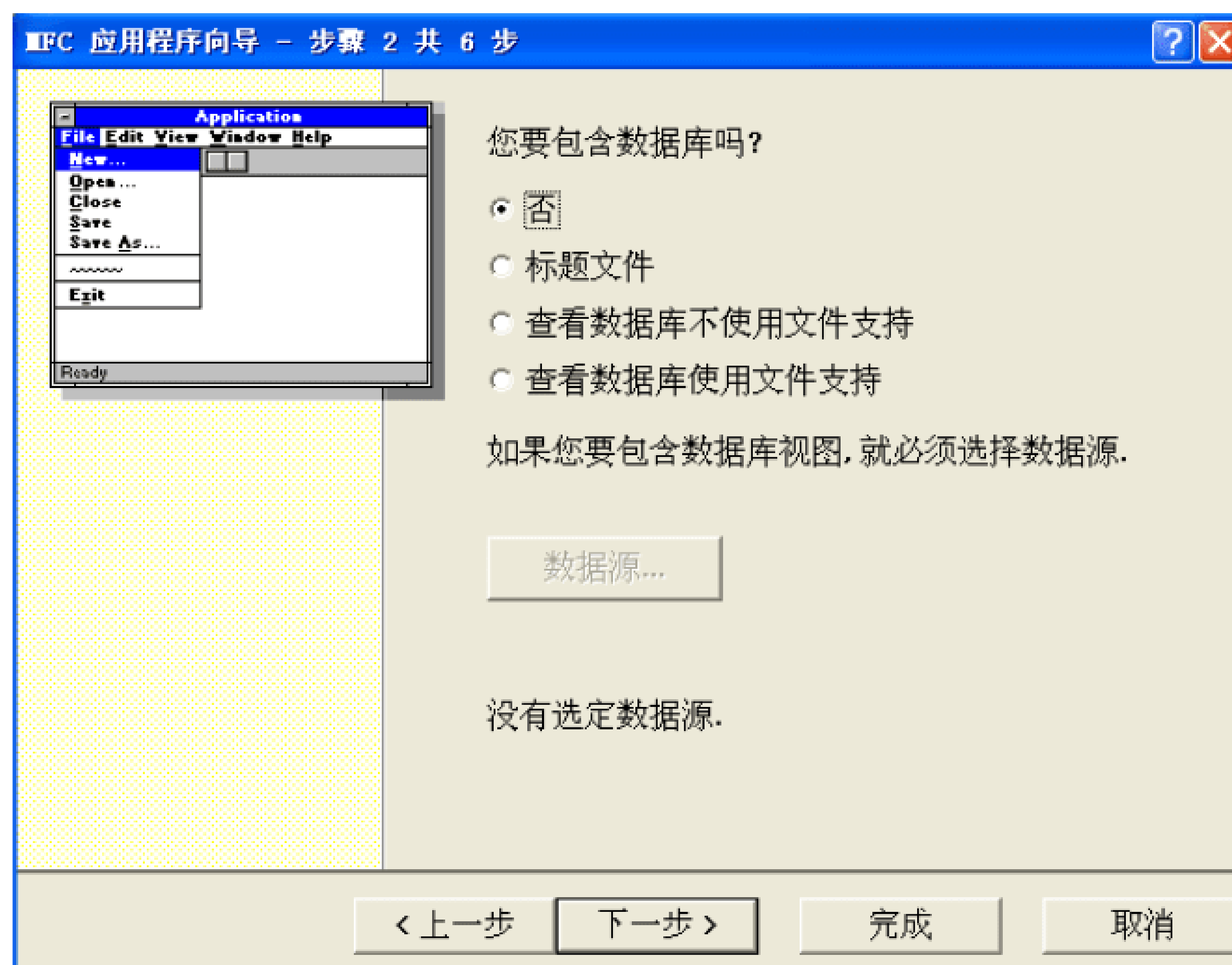


图 10.4　选择是否包含数据库

注意：用户在工程创建的这一步，可以修改工程是否包含数据库的支持。在本实例中，并不包含数据库的支持，所以选择“否”单选按钮。

（4）单击“下一步”按钮，为实例工程选择支持“ActiveX 控件”选项，如图 10.5 所示。

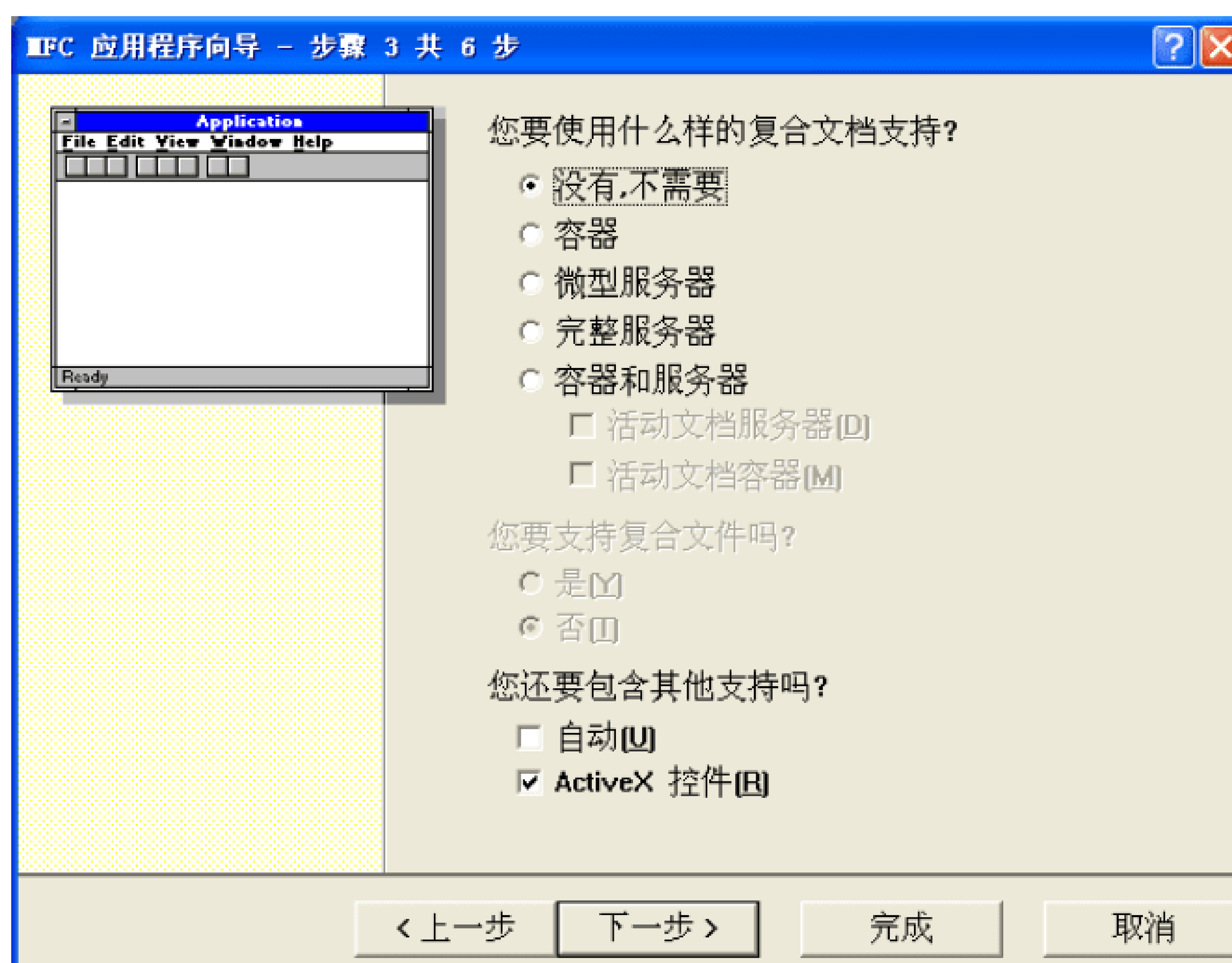


图 10.5　选择“ActiveX 控件”的支持

🔔注意：用户在这一步中，也可以为实例工程同时选择“自动”和“ActiveX 控件”支持。

（5）单击“下一步”按钮，为实例工程添加 Windows Sockets 网络功能的支持，如图 10.6 所示。



图 10.6　为工程选择支持“Windows Sockets”

（6）单击“下一步”按钮，直接来到工程设置步骤的最后一步，修改文档视图类的基类为 CFormView，如图 10.7 所示。

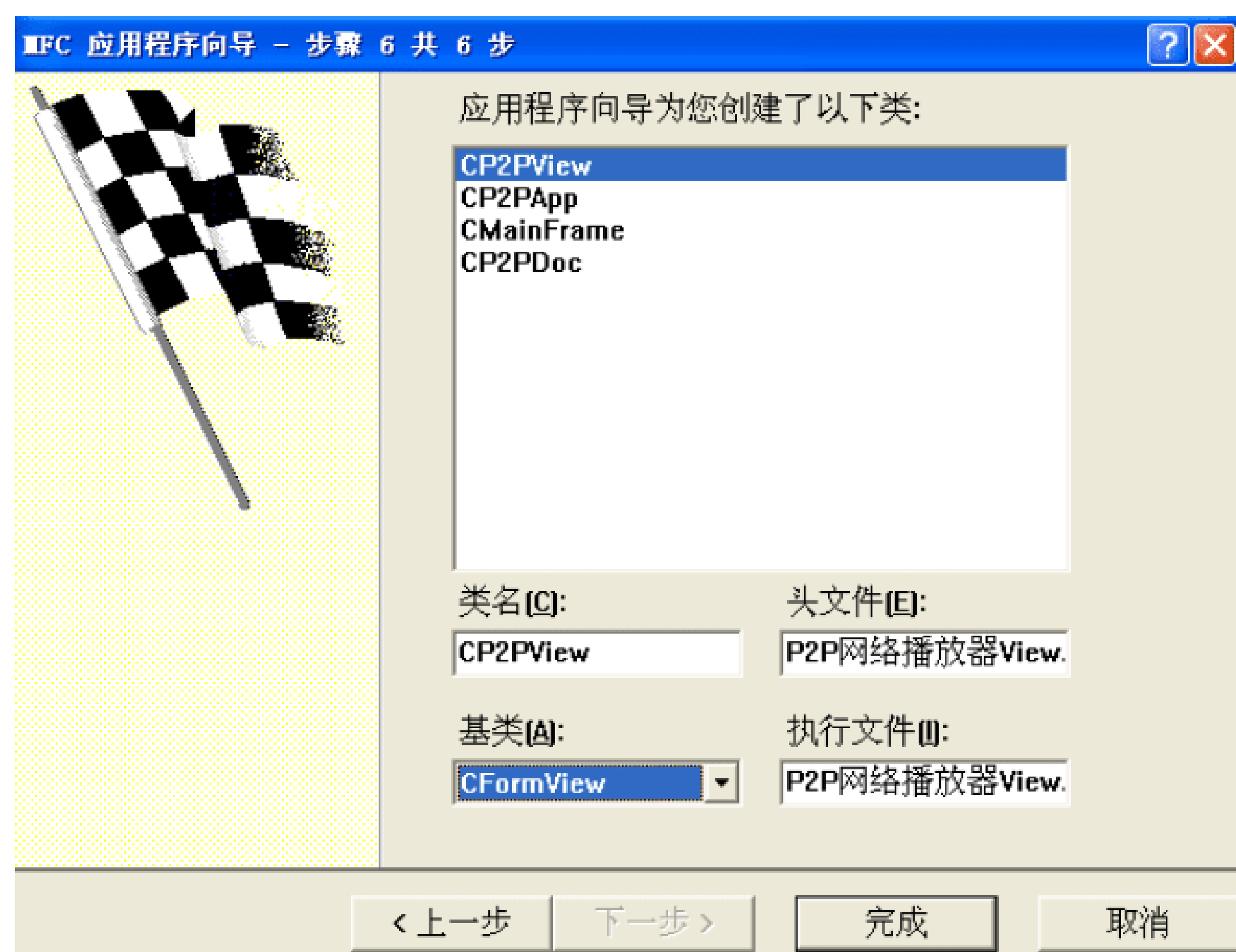


图 10.7　修改实例工程中视图类的基类

（7）用户单击“完成”按钮，成功创建实例工程。这时，用户可以直接运行实例程序，看看工程修改后的效果，如图 10.8 所示。

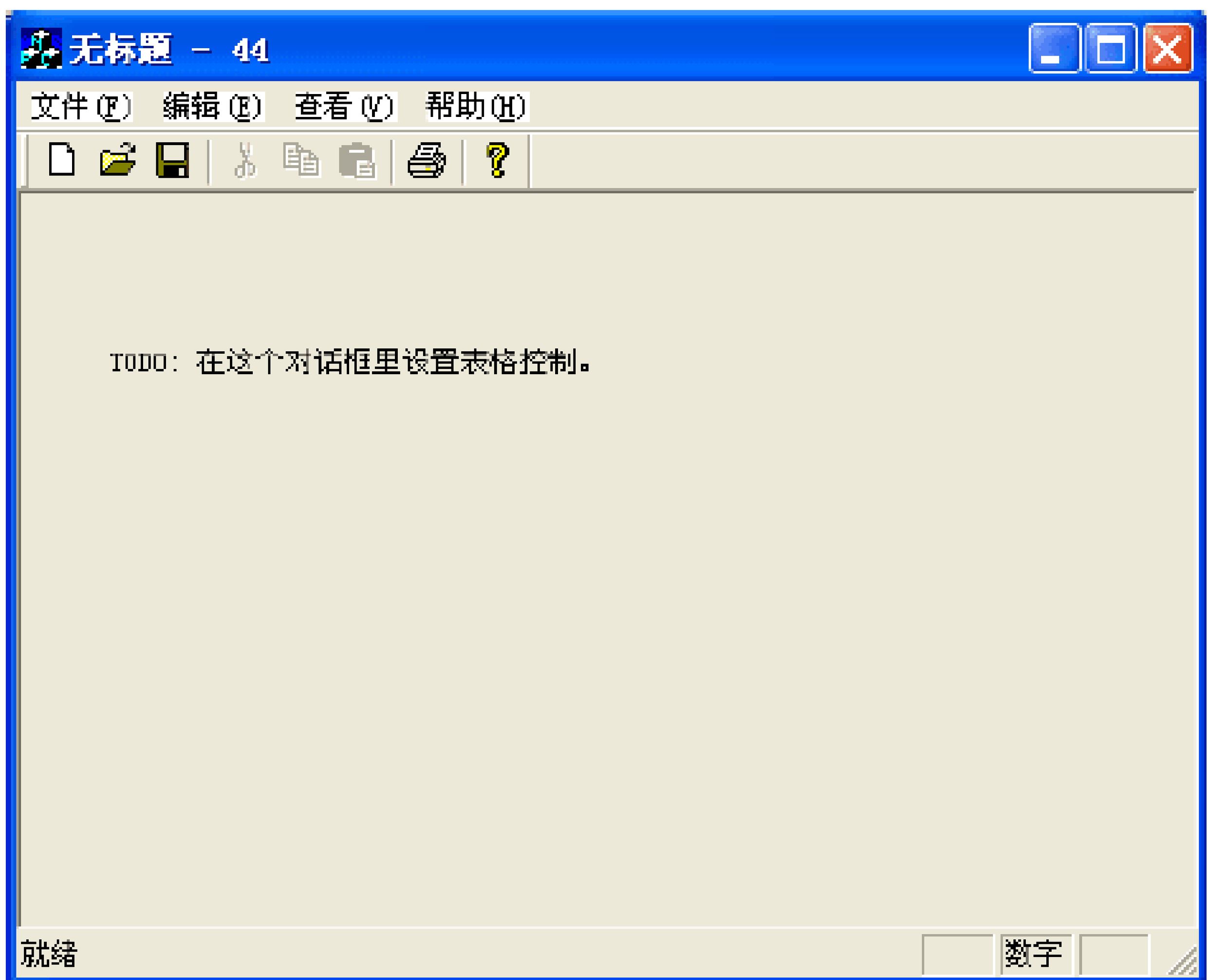


图 10.8　实例程序运行界面

注意：用户在图 10.8 中，可以看到实例运行界面中，单文档的视图变成了对话框视图。这是因为用户在创建工程时，将视图类的基类名设为 CFormView。

10.2.2　界面设计

由于在本实例中，其应用程序类型已经被指定为文档视图类型了。所以，用户在设计界面时，应尽量将界面设计得紧凑。否则，实例程序使用起来显得非常分散，影响实例程序的总体效果。

1．构造程序界面

用户从图 10.8 中可以看到，新建的实例工程界面中并没有任何的功能控件。用户为了使程序实现 P2P 网络播放器的功能，需要为该界面添加相应的功能控件并且需要调整各个控件的大小以及位置达到美化界面的效果。用户添加控件完成后的最终界面，如图 10.9 所示。



图 10.9　界面设计效果

其中，各个控件的 ID 以及作用等，如表 10.1 所示。

表 10.1 控件ID、属性以及作用

控 件 ID	控件属性	控件作用	控 件 ID	控件属性	控件作用
IDC_RADIO1	单选按钮	选择保存网络文件	IDC_KONGZHI	按钮	手动控制播放
IDC_RADIO2	单选按钮	选择删除网络文件	IDC_LIST1	列表控件	显示播放列表
IDC_EDIT1	编辑框	显示网络文件保存路径	IDC_PLAY	按钮	播放
IDC_LIULAN	按钮	选择文件保存的路径	IDC_STOP	按钮	停止
IDC_FIND	按钮	查找网络文件	IDC_PRE	按钮	播放上一首
IDC_FIND2	按钮	查找本地文件	IDC_NEXT	按钮	播放下一首

注意：关于各个控件的实际效果及其功能，请用户参考随书光盘中的代码和其运行效果。

2. 初始化窗口

在本实例中，用户为了保持程序界面的美观，便需要将程序窗口设置为一个固定大小。这个功能可以通过 MFC 函数或者通过改变窗口状态结构体中的相关成员消去窗口中的最大化和最小化按钮。在实例中，将主要选择后者向大家讲解该功能的实现方法。

首先，在 MFC 中，被用于描述窗口状态的结构体是 CREATESTRUCT。用户可以通过改变其成员变量值来改变窗口的显示状态等。其定义如下：

```
typedef struct tagCREATESTRUCT
 {
   LPVOID    lpCreateParams;          //用于创建窗口的数据指针
   HANDLE    hInstance;               //应用程序实例句柄
   HMENU     hMenu;                   //菜单句柄
   HWND      hwndParent;              //指定父窗口类
   int  cy;                           //设置窗口高度
   int  cx;                           //设置窗口宽度
   int  y;                            //指定窗口显示位置的坐标值
   int   x;
   LONG  style;                       //设置窗口的样式
   LPCSTR    lpszName;                //指定窗口的名称
   LPCSTR    lpszClass;               //指定窗口的类名
   DWORD     dwExStyle;               //指定窗口的扩展样式
} CREATESTRUCT;
```

如果用户仅仅需要去掉最大化和最小化按钮，则只需要使用到该结构体中的成员 style 即可。由于本实例是基于文档视图结构，所以，去掉最大化和最小化按钮必须在框架类 CMainFrame 的函数 PreCreateWindow()中实现。代码如下：

```
BOOL CMainFrame::PreCreateWindow(CREATESTRUCT& cs)
{
    if( !CFrameWnd::PreCreateWindow(cs) )
        return FALSE;
cs.style &= ~WS MAXIMIZEBOX;          //去掉最大化按钮
cs.style &= ~WS_MINIMIZEBOX;          //去掉最小化按钮
    return TRUE;
}
```

在代码中，符号“～”表示逻辑运算中的反操作。而在这里表示去掉窗口中的最大化

和最小化按钮，如图 10.10 所示。

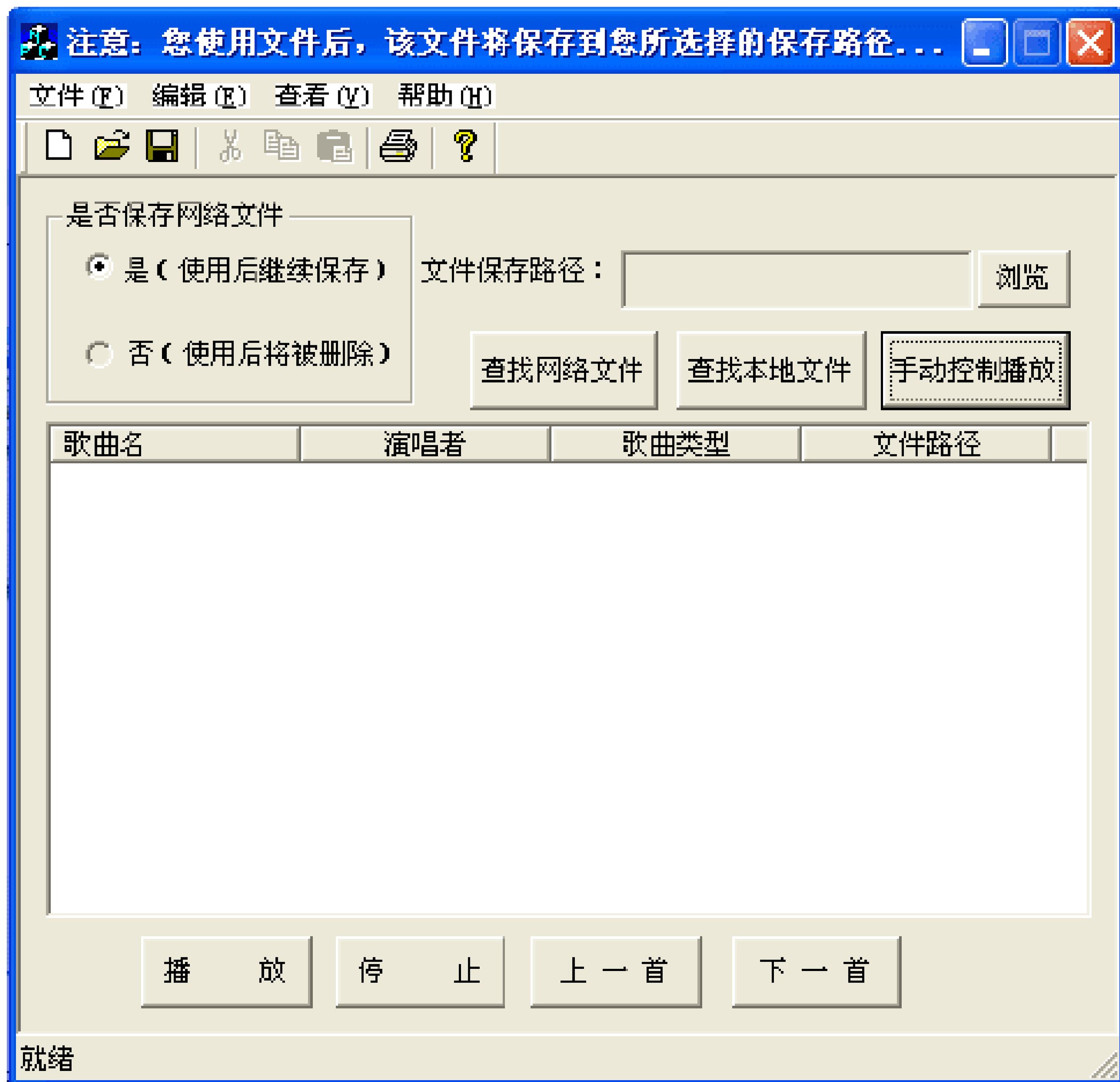


图 10.10　去掉最大化和最小化按钮

注意：由于函数 PreCreateWindow()是框架类创建之前进行调用。所以，用户在修改窗口状态时，一定需要在该函数中进行。用户在实际编程时，如果需要设置该窗口的其他一些初始化状态，例如，窗口大小等，可以通过调用其他函数进行状态设置。这些窗口状态的设置方法将在后面进行讲解。

10.2.3　设置控件初始化状态

在实例程序初始化时，为了使用户造成误操作，影响程序的运行效率。所以，实例窗口显示时需要进行初始化。例如，控件的状态设置以及显示列表控件的标题等信息。

1．初始化列表控件

首先，用户需要按下组合键 Ctrl+W，打开 MFC 应用程序向导对话框并为列表控件关联一个对应的控件变量，如图 10.11 所示。

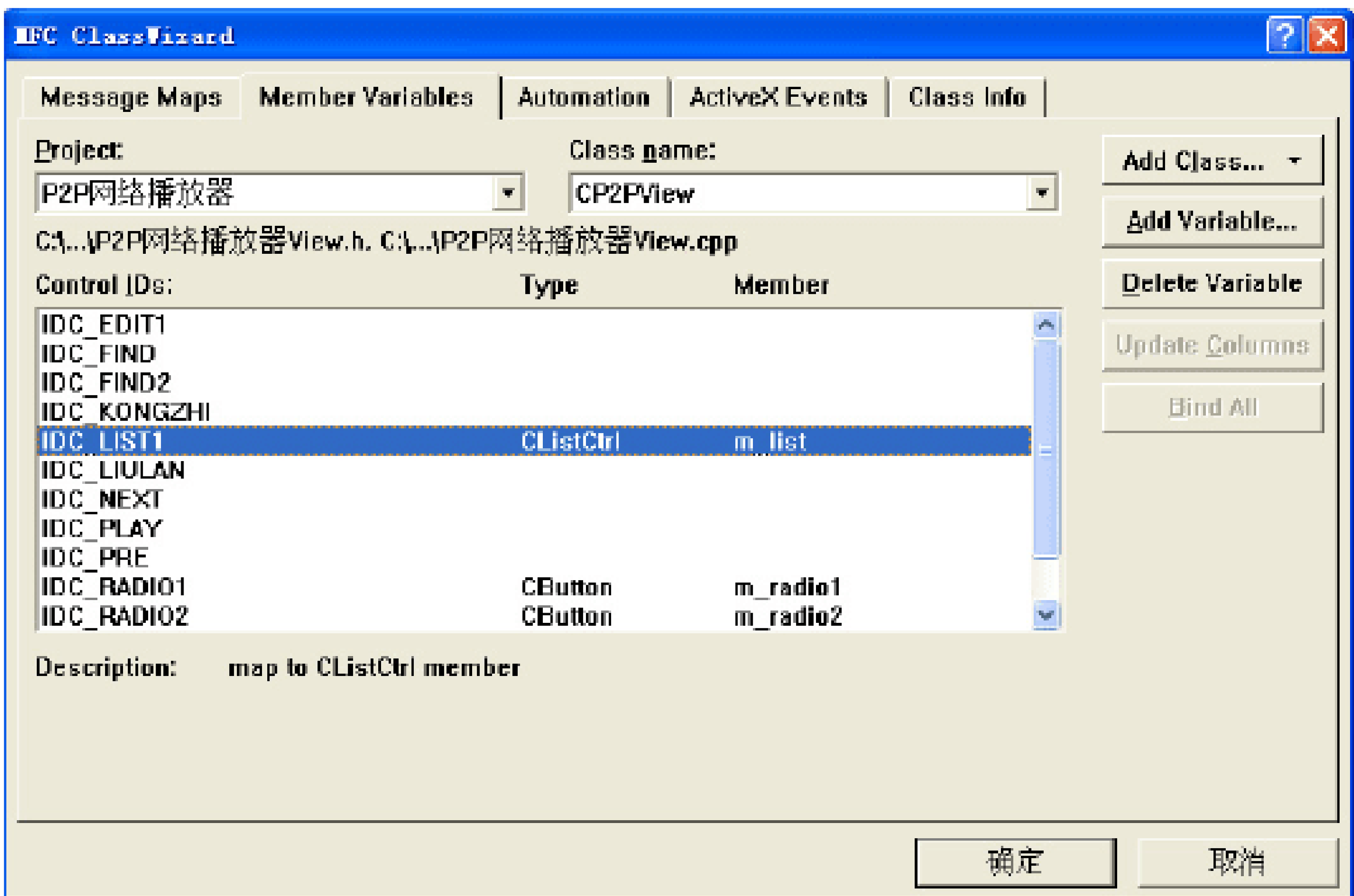


图 10.11　打开 MFC 向导中的 Member Variables 选项卡

用户可以在控件列表中选择操作控件的 ID 号码即可。在这里，用户需要选择的控件 ID 是 IDC_LIST1，再单击 Add Variables 按钮为刚选择的控件添加相应的控件变量，如图 10.12 所示。

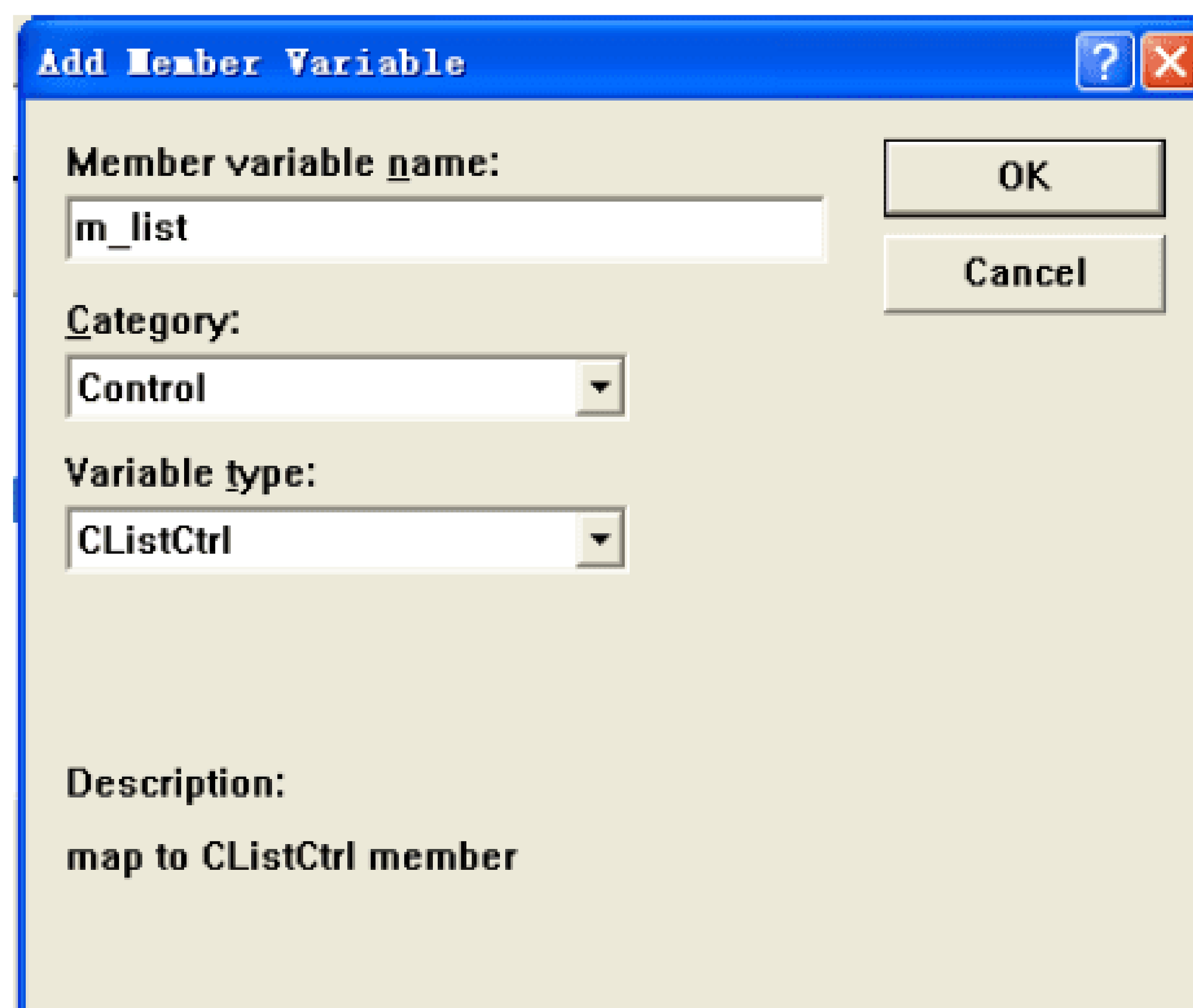


图 10.12　添加控件变量

用户在第一个编辑框中输入添加的控件变量名 m_list，其他则默认。单击 OK 按钮，返回 VC 主界面。用户可以在工程视图类中看到刚添加的列表控件变量。代码如下：

```
class CP2PView : public CFormView              //视图类
{
protected:
    CP2PView();
    DECLARE_DYNCREATE(CP2PView)
public:
...                                            //省略部分代码
    CListCtrl   m_list;                        //列表控件变量
   }
```

然后，在函数 OnInitialUpdate()中进行列表控件的初始化。代码如下：

```
void CP2PView::OnInitialUpdate()
{
CFormView::OnInitialUpdate();               //调用基类的函数
LVCOLUMN lv;                                //定义列表控件的结构体变量
lv.mask=LVCF_TEXT|LVCF_FMT|LVCF_WIDTH;      //初始化该结构体中的各个成员
lv.fmt=LVCFMT_CENTER;
lv.pszText="歌曲名";                        //设置第一列的标题
lv.cx=110;                                  //设置第一列的宽带
m_list.InsertColumn(0,&lv);                 //插入第一列
lv.pszText="演唱者";                        //设置第二列的标题
lv.cx=105;                                  //设置第二列的宽度
m_list.InsertColumn(1,&lv);                 //插入第二列
lv.pszText=""歌曲类型";
lv.cx=100;
m_list.InsertColumn(2,&lv);
lv.pszText="文件路径";
lv.cx=110;
m list.InsertColumn(3,&lv);
...                                         //省略部分代码
}
```

在上面的代码中，用户主要是使用了结构体 LVCOLUMN 的变量 lv 对即将插入的各列信息进行描述。再通过列表控件的成员函数 InsertColumn()插入该列。将代码保存以后，进行编译运行，结果如图 10.13 所示。

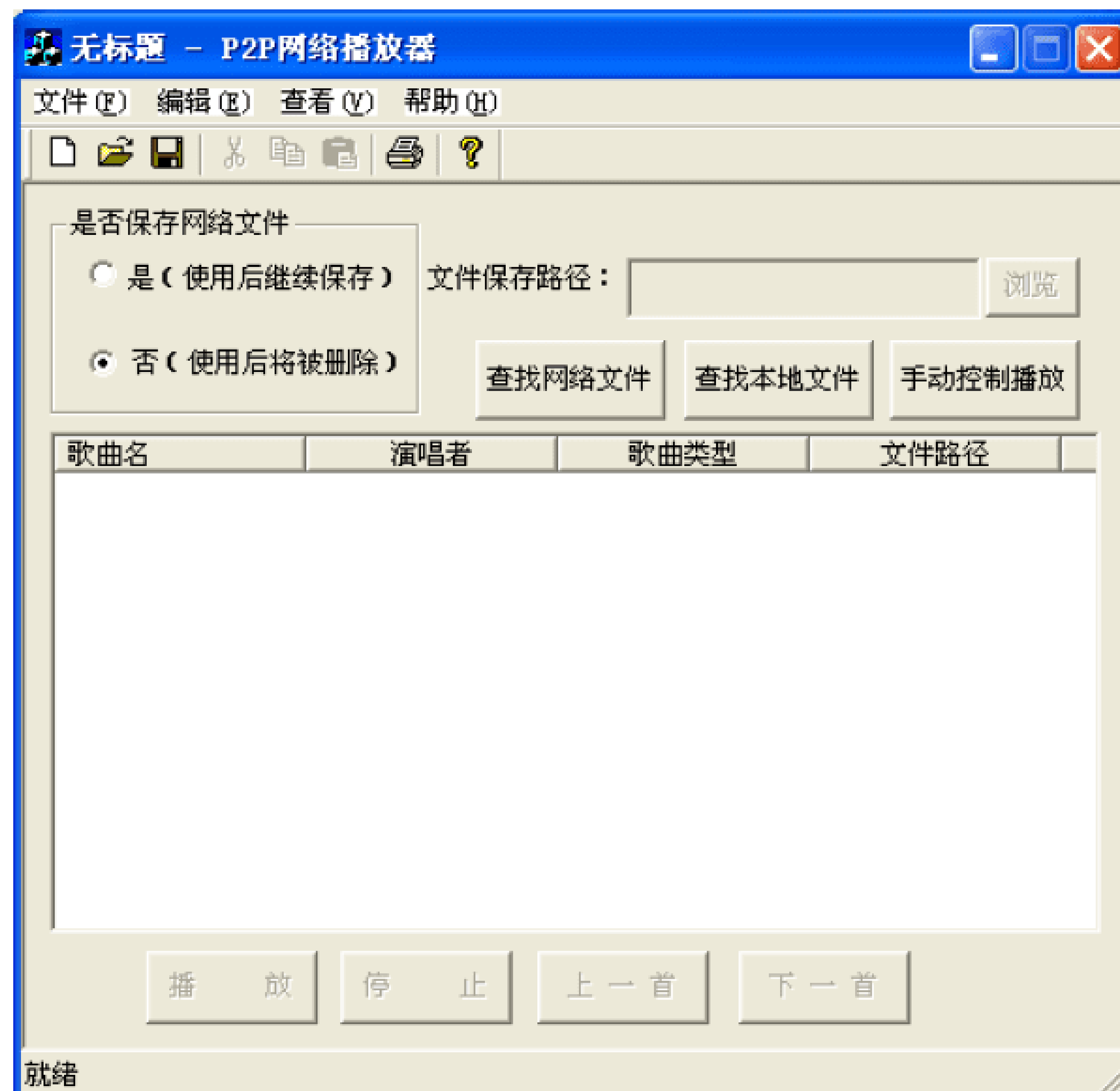


图 10.13　为列表控件成功添加标题

注意： 函数 OnInitialUpdate()是视图类中更新界面时所调用的函数。所以，各个控件的初始化状态设置都必须在该函数中进行。而结构体 LVCOLUMN 是 MFC 中专门用于描述列表控件中各列的相关信息。

2. 初始化其他控件

在程序中，某些按钮必须等到一定的时间才会被使用。所以，在程序初始化时，用户需要将这些按钮禁用。代码如下：

```
void CP2PView::OnInitialUpdate()
{
    CFormView::OnInitialUpdate();
...                                              //省略部分代码
m_radio1.SetCheck(1);                            //设置单选按钮的选择状态
GetDlgItem(IDC_EDIT1)->EnableWindow(false);      //禁用文件路径编辑框
GetDlgItem(IDC_LIULAN)->EnableWindow(false);     //禁用浏览按钮
GetDlgItem(IDC_PLAY)->EnableWindow(false);       //禁用播放按钮
GetDlgItem(IDC_STOP)->EnableWindow(false);       //禁用停止按钮
GetDlgItem(IDC_PRE)->EnableWindow(false);        //禁用上一首按钮
GetDlgItem(IDC_NEXT)->EnableWindow(false);       //禁用下一首按钮
}
```

代码运行后，窗口中的相应按钮均处于被禁用状态，如图 10.14 所示。



图 10.14　窗口界面初始化状态

当用户选择保存文件后，文件保存路径编辑框和浏览按钮会变为可用状态。而当播放列表中有记录时，则播放、停止、上一首和下一首按钮也会变为可用状态。

10.2.4　添加消息响应函数

用户在程序中需要实现真正的功能，必须首先为各个控件添加相应的消息响应函数。然后，在这些消息响应函数中编写代码实现这些功能。

在 VC 中，用户可以通过 MFC 应用程序向导为控件添加相应的消息响应函数。例如，用户为浏览按钮添加单击消息的响应函数，如图 10.15 所示。

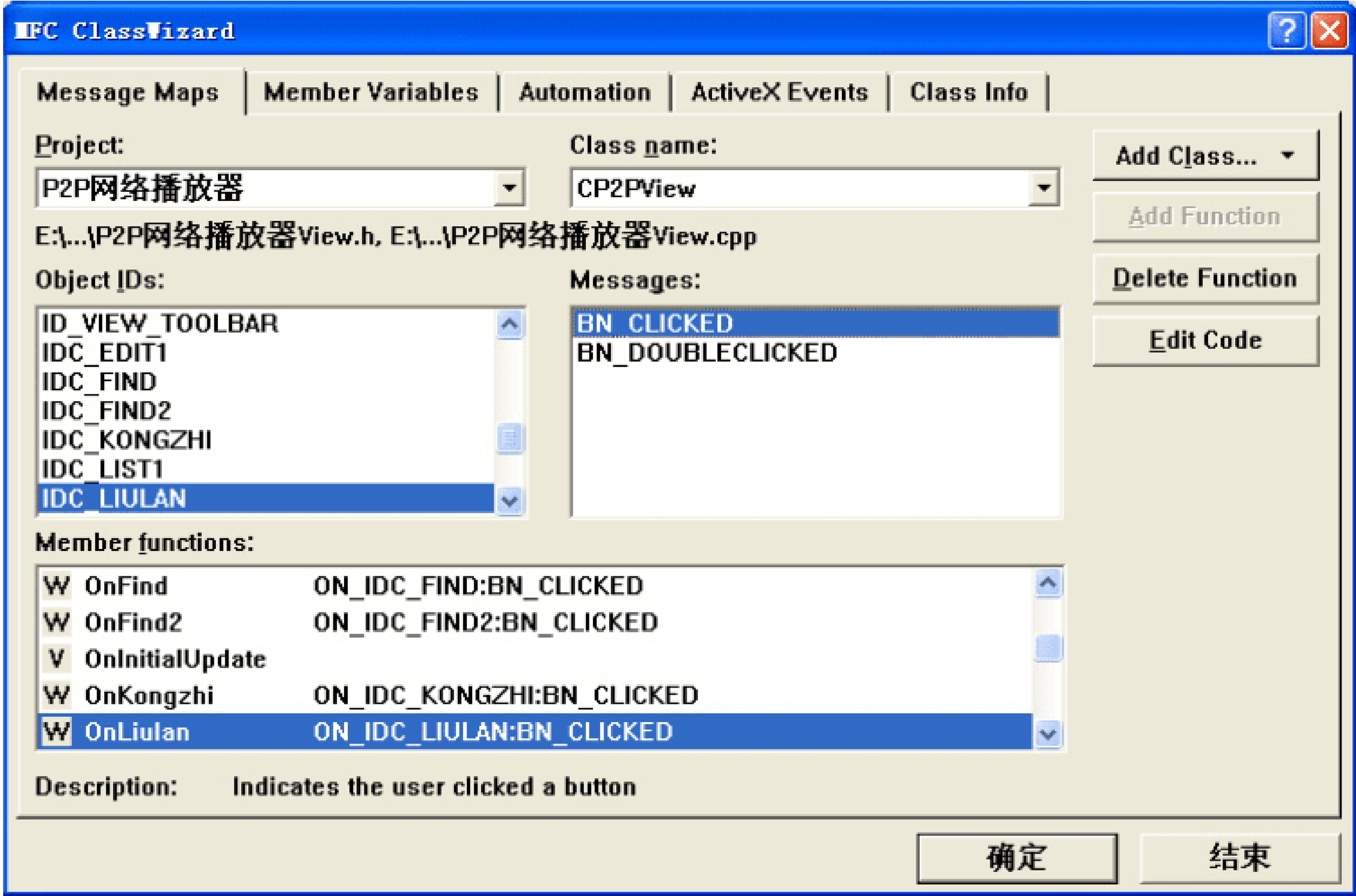


图 10.15　为浏览按钮添加消息响应函数

用户在控件 ID 列表中选择浏览按钮的 ID 后，在 Messages 列表中选择单击消息 BN_CLICKED。然后单击 Add Function 按钮，弹出 Add Member Function 对话框，如图 10.16 所示。

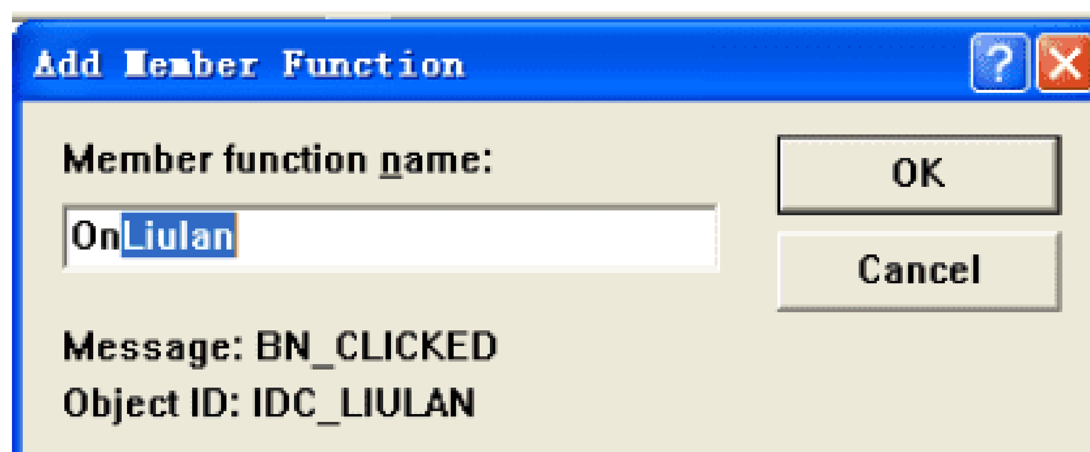


图 10.16　Add Member Function 对话框

用户在 Add Member Function 对话框中，可以修改消息响应函数的名称。在实例中，该消息响应函数名为 OnLiulan()。修改函数名完成之后，单击 OK 按钮，完成消息响应函数的添加。

用户在 VC 环境下，为控件添加消息响应函数，还可以在对话框面板上，使用鼠标左键双击对应控件，也会弹出如图 10.16 所示的 Add Member Function 对话框。

注意：本实例中，为控件添加消息响应函数的方法相同。所以，用户可以按照上面所介绍的为浏览按钮添加单击消息响应函数的方法及步骤，为其他控件添加消息响应函数。本小节不再进行赘述。

10.2.5　向播放列表添加 MP3 文件

在 10.2.4 节中，用户为实例中的所有控件均添加了响应的控件消息响应函数。因此，用户需要在这些消息响应函数中添加代码，以实现向播放列表中添加 MP3 文件的功能。由于该实例程序具有查找本地文件和查找网络文件的功能，所以，在本节中将首先向用户介绍查找和播放本地 MP3 文件的实现。而网络文件方面的操作将在 10.3 节中向用户进行讲解。

1．实现组框选择功能

当程序开始运行时，用户需要在“是否保存网络文件”组框中，选择是否保存当前所使用的 MP3 文件。如果用户选择“否”单选按钮，则“浏览”按钮处于禁用状态。否则，“浏览”按钮将处于可用状态，以便用户选择文件的保存路径。代码如下：

```
void CP2PView::OnRadio1()                            //用户选择“是”单选按钮
{
    GetDlgItem(IDC_LIULAN)->EnableWindow(true);  //使“浏览”按钮处于可用状态
    GetDlgItem(IDC_RADIO2)->EnableWindow(false); //禁用组框中的“否”单选按钮
::SetWindowText(this->GetParent()->m_hWnd,"注意：文件使用后将保存到您所选择的
保存路径中！");
                                                     //设置框架标题，提醒用户
}
void CP2PView::OnRadio2()                            //用户选择“否”单选按钮
{
```

```
    ::SetWindowText(this->GetParent()->m_hWnd,"注意：您选择了使用文件后，将其
    删除！");
                                                    //设置框架标题，提醒用户
    }
```

用户在函数 OnRadio1()中，实现了“浏览”按钮的状态改变，并且禁用了“否”单选按钮，防止用户误操作，引起程序运行异常。而在函数 OnRadio2()中，只是简单地设置了提醒消息，并没有“是”单选按钮，这样可以方便用户保存文件操作。程序运行后，用户可以分别选择组框中的两个单选按钮，如图 10.17 和图 10.18 所示。

用户在图 10.17 中，可以看到当选择了“是”单选按钮后，“浏览”按钮处于可用状态，并且在标题上出现了提示信息。

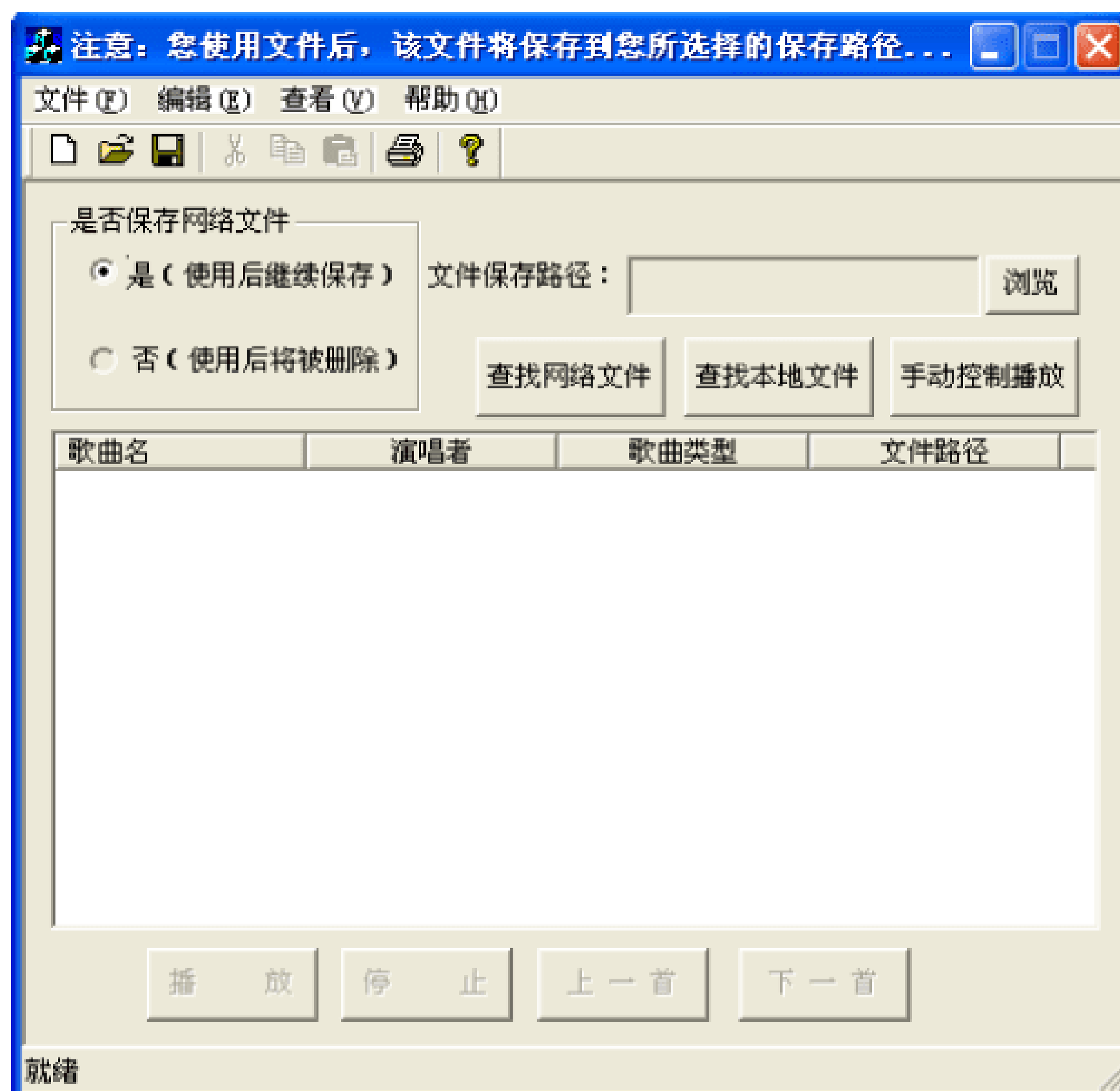


图 10.17　选择“是”单选按钮改变界面显示

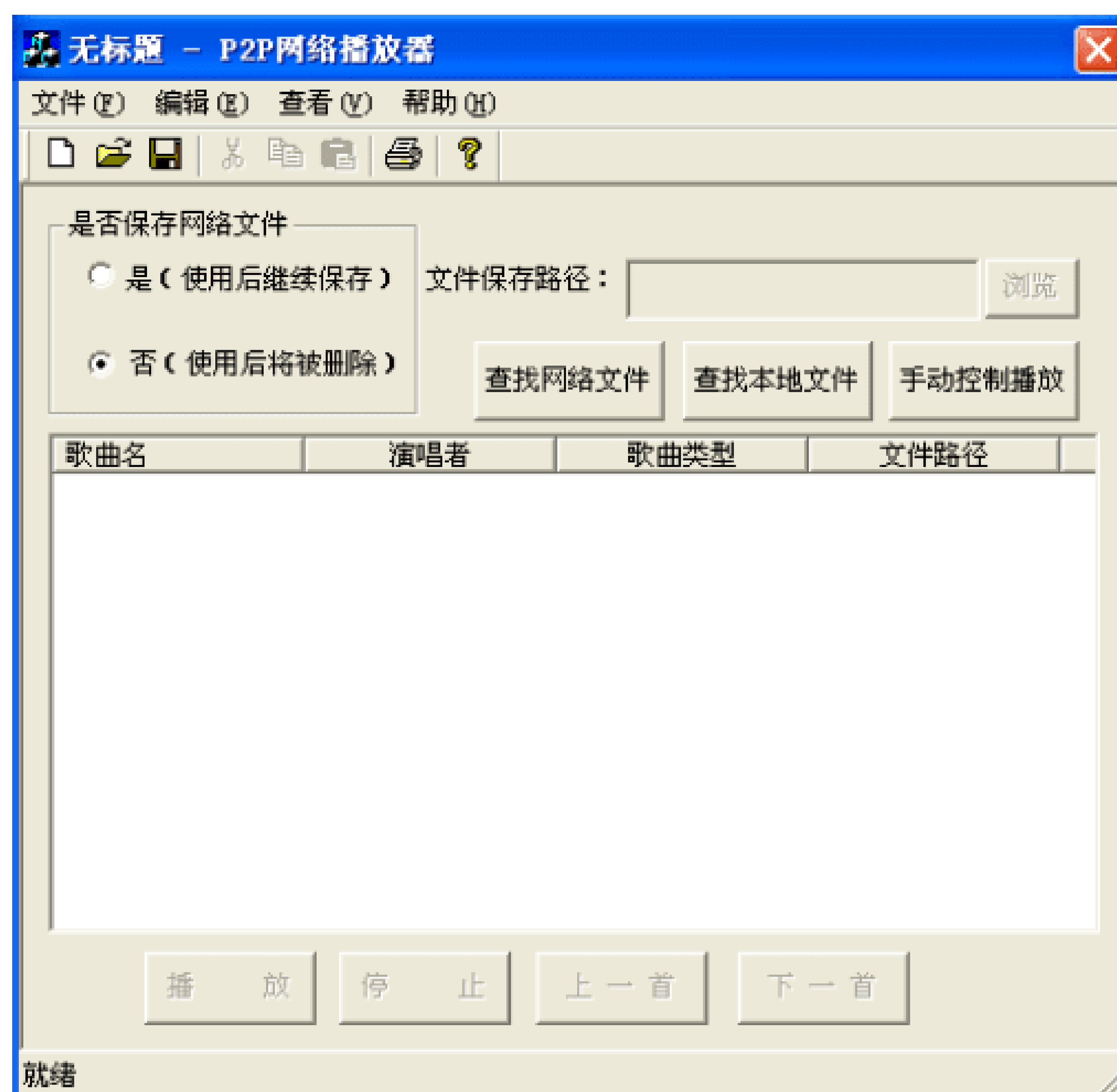


图 10.18　选择“否”单选按钮出现提示信息

在组框中，用户主要实现是否保存当前所使用文件的功能。如果需要保存，则选择所保存的路径。否则，提示用户当前选择的选项功能。

2. 实现浏览按钮功能

当用户在组框中选择了“是”单选按钮后，需要通过该按钮选择文件的保存路径，并将该路径显示在路径编辑框中。代码如下：

```
void CP2PView::OnLiulan()
{
    CString str2,strTmp;                              //定义字符串变量
    BROWSEINFO  bBinfo;                               //定义结构变量
   memset(&bBinfo,0,sizeof(BROWSEINFO));              //定义结构并初始化为 0
   bBinfo.hwndOwner=m_hWnd;                           //设置对话框所有者句柄
           bBinfo.lpszTitle="请选择保存路径：";       //修改显示的字符串
           bBinfo.ulFlags=BIF_RETURNONLYFSDIRS;       //设置标志只允许选择目录
           LPITEMIDLIST lpDlist;
   lpDlist=SHBrowseForFolder(&bBinfo)   ;             //显示目录选择对话框
   if(lpDlist!=NULL)
   {
     SHGetPathFromIDList(lpDlist,strTmp.GetBuffer(0));
                                            //把项目标识列表转化成目录字符串
        GetDlgItem(IDC_EDIT1)->SetWindowText(strTmp);
                                            //将路径字符串显示在编辑框中
}
```

在代码中，用户首先调用函数 SHBrowseForFolder()弹出浏览文件夹对话框。然后，调用函数 SHGetPathFromIDList()将用户的选择转换为路径字符串显示在编辑框中。

注意：函数 SHBrowseForFolder()用于显示“浏览文件夹”对话框。其参数是指向结构体 BROWSEINFO 的指针变量，用于设置该“浏览文件夹”对话框的基本信息。在本节中，不对该函数进行详细的介绍，用户只需明白其功能即可。

当用户单击“浏览”按钮以后，程序会弹出“浏览文件夹”对话框，如图 10.19 所示。用户在“浏览文件夹”对话框中选择好文件所保存的路径后，单击“确定”按钮返回到程序主界面。此时，在文件保存路径文本框中已经显示了用户刚才选择的保存路径，如图 10.20 所示。



图 10.19　“浏览文件夹”对话框

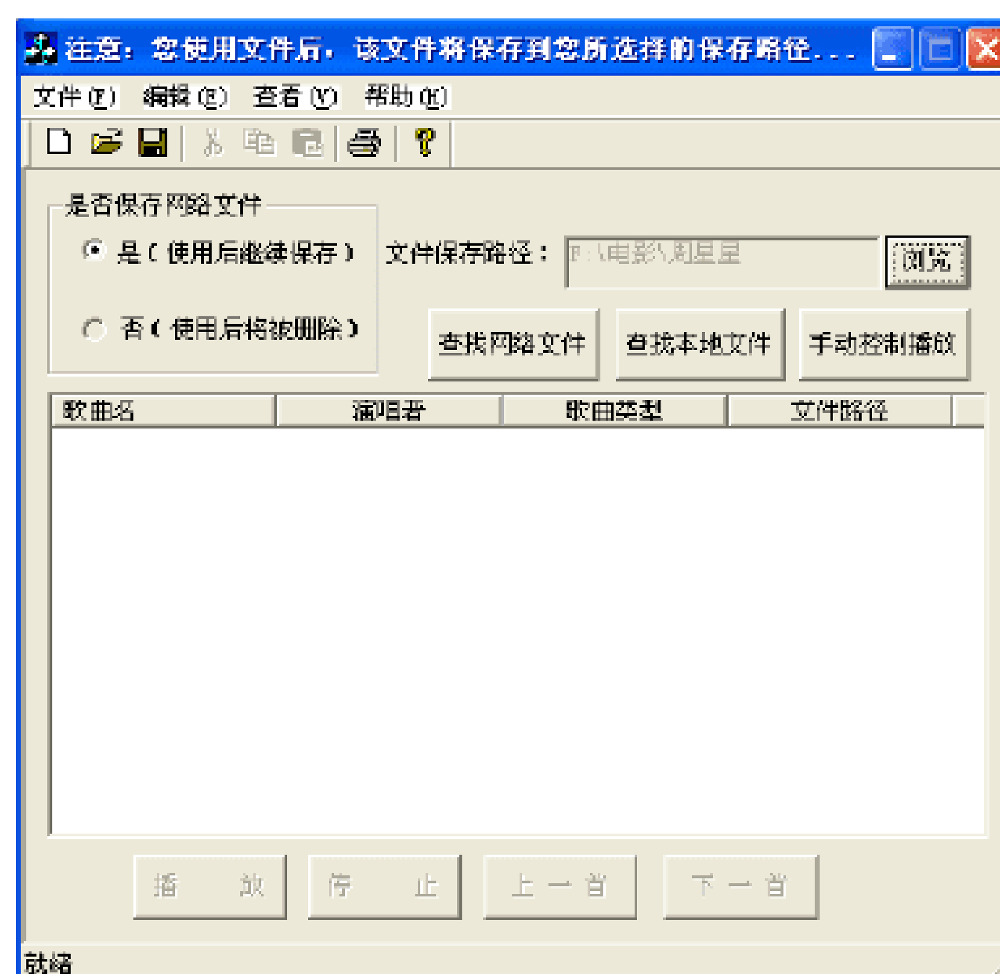


图 10.20　显示所选择的文件保存路径

在这一节中，主要向用户介绍了显示“浏览文件夹”对话框的方法。

3．实现查找本地文件功能

在本章实例中，用户不但可以访问网络中的相关文件，还可以搜索本地相关的文件。然后，将搜索到的文件添加到列表中。这个功能是在查找本地文件按钮的消息响应函数中实现，其代码如下：

```
void CP2PView::OnFind2()
{
CFileFind findfile;                                  //定义搜索文件类对象
CString str2,strTmp;                                 //定义字符串
CString strpath="";
BROWSEINFO  bBinfo;                                  //定义结构体变量
    memset(&bBinfo,0,sizeof(BROWSEINFO));            //定义结构并初始化
    bBinfo.hwndOwner=m_hWnd;                         //设置对话框所有者句柄
            bBinfo.lpszTitle="请选择安装路径：";       //修改显示的文字信息
            bBinfo.ulFlags=BIF_RETURNONLYFSDIRS;     //设置标志只允许选择目录
            LPITEMIDLIST lpDlist;
    lpDlist=SHBrowseForFolder(&bBinfo)   ;           //显示选择对话框
    if(lpDlist!=NULL)
    {
    SHGetPathFromIDList(lpDlist,strTmp.GetBuffer(0));//把选项转化成目录字符串
strcat(strTmp.GetBuffer(0),"\\*.mp3");               //连接字符串
findfile.FindFile(strTmp);                           //查找指定类型的文件
while(findfile.FindNextFile())                       //继续查找文件
{
str2=findfile.GetFilePath();                         //获取文件路径
CFile file(str2,CFile::modeRead);                    //创建文件对象
file.Seek(-128,CFile::end);                          //从文件结尾处移动文件指
针
file.Read(&mp.mp3,128);                              //读取文件
file.Close();                                        //关闭文件
int nRow=m_list.InsertItem(m_list.GetItemCount()+1,mp.mp3.title);//插入行
m_list.SetItemText(nRow,1,mp.mp3.arti);              //设置列表数据
if(mp.mp3.heade && "TAG")                            //如果文件是 MP3 格式
{
    CString mp3="MP3";                               //定义字符串
m_list.SetItemText(nRow,2,mp3);                      //设置列表数据
}
m_list.SetItemText(nRow,3,str2);
sprintf(mp.str,"%s",str2.GetBuffer(1));              //格式化输出字符数组
mp.mp3.text[28]=0;                                   //填充结构体中的各个成员
mp.mp3.year[4]=0;
mp.mp3.alb[30]=0;
CFile file1("列表.TXT",CFile::modeReadWrite|CFile::modeNoTruncate
|CFile::modeCreate|CFile::typeBinary);           //创建文件对象
file1.Seek(0,CFile::end);                        //将文件指针移动到文件最后
file1.Write(&mp,sizeof(mp));                     //将数据写入文件
file1.Flush();                                   //强制写入数据
file1.Close();                                   //关闭文件
}
}
}
```

在代码中，用户仍然是调用函数 SHBrowseForFolder()显示“浏览文件夹”对话框。程序根据用户在该对话框上的选择，返回所选目录的路径。接着，程序将该路径与字符串“..*.mp3”连接，表示搜索该路径下所有 MP3 文件。

注意：在程序中，结构体变量 mp 是用户在第 9 章中自定义结构体 mp3str 的结构体变量。

搜索 MP3 文件的功能主要由文件搜索类 CFileFind 完成。当用户搜索到相应文件时，使用文件对象从文件数据中读取 128 字节的数据到结构体 mp3str 中。程序在每读取一个 MP3 文件时，便将该文件的相关信息添加到播放列表并写入文件中保存。用户运行程序，单击“查找本地文件”按钮，如图 10.21 所示。

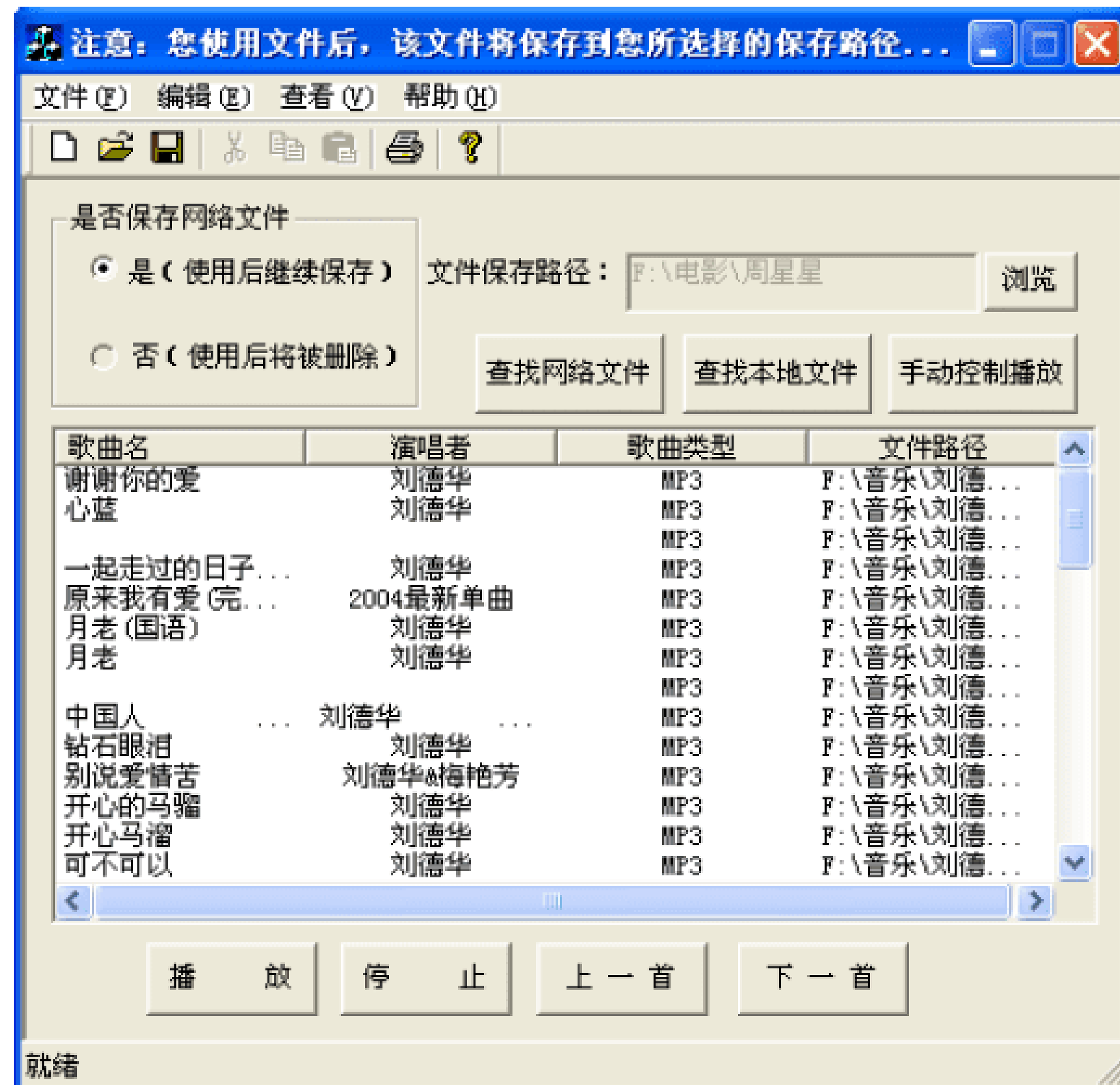


图 10.21　搜索本地文件

用户可以从图 10.21 中看到，搜索到的 MP3 文件都被添加到了列表中。

注意：在本节中，由于“查找网络文件”按钮的消息响应函数涉及的知识很多，需要向用户详细讲解。所以，对于“查找网络文件”按钮的消息响应函数实现方法将在后面几节中讲解。

10.2.6　播放 MP3 文件

在程序中，用户播放 MP3 文件可以通过双击播放列表中的选项进行播放，或者是通过手动控制进行 MP3 文件的播放以及控制。在本节中，将向用户介绍这两种方法的具体实现。

1．双击列表选项播放MP3

当用户通过搜索功能将 MP3 文件添加到列表中，双击列表选项后，程序应该播放对应的 MP3 文件。该功能与第 9 章中的功能实现基本一致，均是通过调用 MCI 函数实现。首先，为列表控件添加鼠标双击消息响应函数。然后，在该响应函数中添加代码。其代码

如下：

```
void CP2PView::OnDblclkList1(NMHDR* pNMHDR, LRESULT* pResult)
{
    CString str;                                           //定义字符串变量
POSITION pos=m_list.GetFirstSelectedItemPosition();    //获取用户双击位置
if(pos==NULL)                                          //判断位置是否为空
{
    MessageBox("用户双击位置错误！");                      //显示错误信息
}
else
{
        int nItem=m_list.GetNextSelectedItem(pos);//获得双击位置的索引值
        index=nItem;
str=m_list.GetItemText(nItem,3);                  //获得该索引位置的第三列字符
if(open.wDeviceID)                                //判断 MCI 设备是否初始化
{
  mciSendCommand(open.wDeviceID,MCI_CLOSE,0,0);//关闭 MCI 设备
}
open.lpstrElementName=str;                        //向 MCI 设备传送文件路径
open.lpstrDeviceType="mpegvideo";                 //指定 MCI 设备类型
err=mciSendCommand(0,MCI_OPEN,MCI_OPEN_TYPE|MCI_OPEN_ELEMENT|MCI_WAIT,(
DWORD)(LPVOID)&open);                             //初始化 MCI 设备
if(err==0)                                        //如果设备初始化成功
{
MCI_PLAY_PARMS play;                              //定义播放结构体
mciSendCommand(open.wDeviceID,MCI_PLAY,0,(DWORD)&play);
                                                  //向 MCI 设备发送播放指令
}
}
    *pResult = 0;
}
```

在程序中，变量 err、index、open 等均为视图类中定义的成员变量。代码如下：

```
class CP2PView : public CFormView
{
public:
    CP2PDoc* GetDocument();
    mp3str mp;
    MCI_OPEN_PARMS open;                          //MCI 设备初始化结构体
    DWORD err;                                    //错误信息变量
    int index;                                    //记录当前播放列表的索引值
   ...                                            //省略部分代码
  }
```

注意： 用户在实际编程时，一定需要首先判断指定 ID 号的 MCI 设备是否存在。如果存在，则向该设备发送关闭指令以后，再重新进行初始化。否则用户编写的程序不能使用该 MCI 设备。如果用户对 MCI 函数的一些用法还不熟悉，请复习第 9 章中的相关内容或结合实例程序与 MSDN 进行学习。

2. 改变播放控制按钮的状态

用户在实例程序中，除了使用双击列表选择的方法播放 MP3 以外，还可以使用手动控制的方法进行播放。但是，在程序初始化时，这些播放控制按钮均处于禁用状态。因此，

用户必须在程序中修改其状态设置，实现这一功能可以有两种方法。分别是通过“手动控制播放”按钮和视图显示更新函数 OnDraw()。下面将向用户介绍这两种方法的代码实现。

首先，在“手动控制播放”按钮的消息响应函数 OnKongzhi()中，改变“播放”等控制按钮的显示状态。代码如下：

```
void CP2PView::OnKongzhi()
{
GetDlgItem(IDC_PLAY)->EnableWindow(true);       //改变各个控件的显示状态
GetDlgItem(IDC_STOP)->EnableWindow(true);
GetDlgItem(IDC_PRE)->EnableWindow(true);
GetDlgItem(IDC_NEXT)->EnableWindow(true);
flag=1;                                         //设置状态标志
}
```

用户将上面的代码编译运行。然后单击“手动控制播放”按钮，会看到几个控制播放的按钮均处于可用状态，如图 10.22 所示。



图 10.22　使用“手动控制播放”按钮改变控制按钮的状态

用户在实际编程中，除了使用“手动控制播放”按钮改变“播放”等主要控制按钮的状态以外。还可以根据播放列表中是否有可用的 MP3 文件，对这些主要控制按钮的状态进行设置。代码如下：

```
void CP2PView::OnDraw(CDC* pDC)
{
    if(m_list.GetItemCount()!=0 && flag!=1)//判断播放列表中的文件数目是否为 0
    {                               //若数目不为 0，则将控制按钮的状态设置为可用
GetDlgItem(IDC_PLAY)->EnableWindow(true);
GetDlgItem(IDC_STOP)->EnableWindow(true);
GetDlgItem(IDC_PRE)->EnableWindow(true);
GetDlgItem(IDC_NEXT)->EnableWindow(true);
    }
}
```

用户将以上程序编译运行后，单击“搜索本地文件”按钮向播放列表中添加 MP3 文件。然后，用户会看到“播放”等控制按钮均处于可用状态。这是因为当用户向列表中添加文件成功后，程序界面会使用函数 OnDraw()进行更新，如图 10.23 所示。

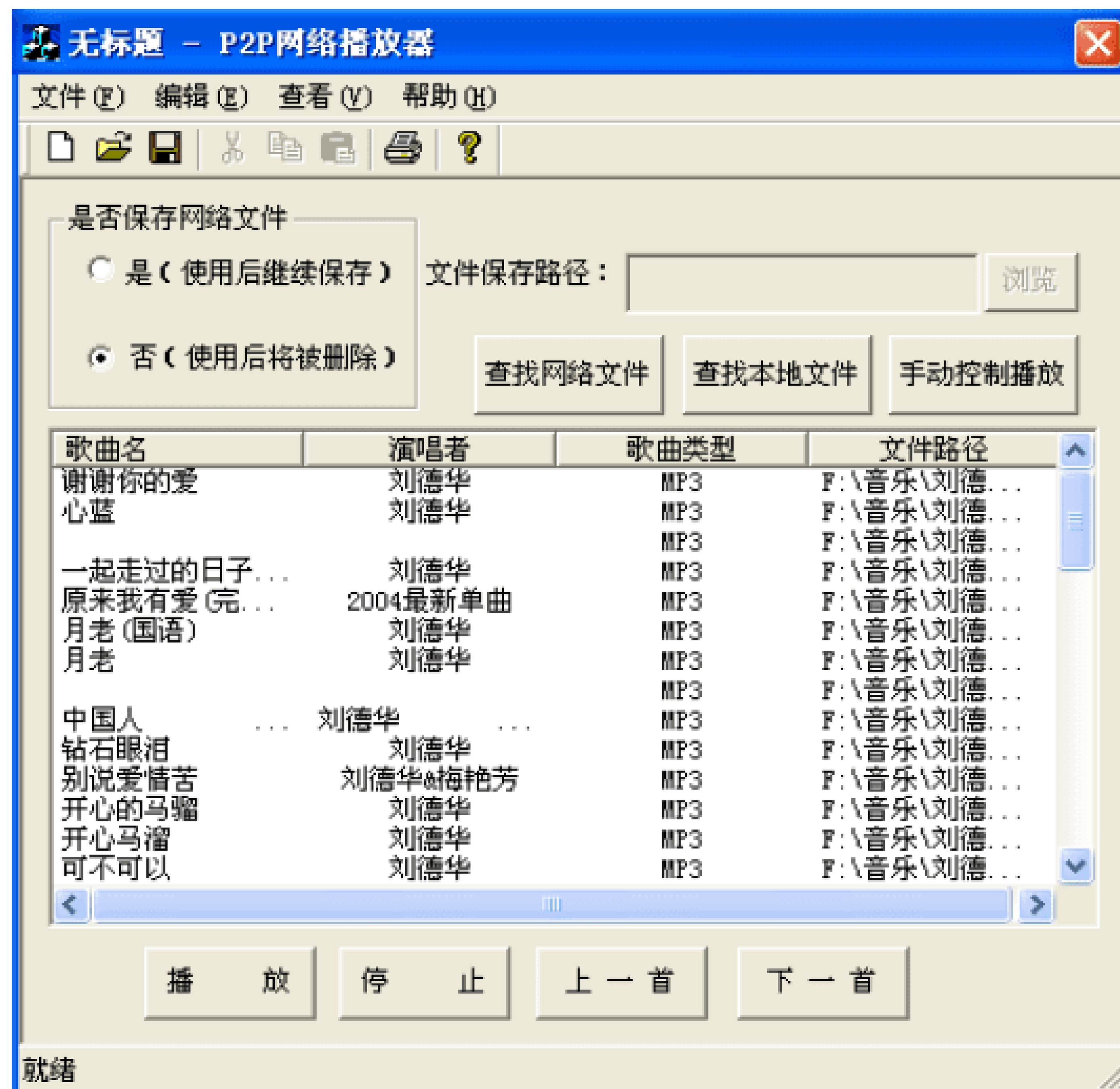


图 10.23　使用视图更新函数对控制按钮的状态进行修改

当这些主要控制播放的按钮可用后，用户便可以使用这些按钮来对 MP3 文件的播放进行控制。例如，暂停、播放、更换曲目等操作。

3．实现播放控制

当播放控制按钮处于可用状态后，用户便可以使用这些按钮控件对 MP3 文件的播放状态进行相关控制。首先，用户在“播放”按钮的消息响应函数 OnPlay()中，添加播放控制的相关代码，其代码如下：

```
void CP2PView::OnPlay()
{
    CString str,text;                           //定义字符串变量
    char str1[100];                             //定义字符数组
    str=m_list.GetItemText(n,3);                //获取默认列表项中的 MP3 文件路径
if(open.wDeviceID)                              //判断指定 ID 的 MCI 设备是否初始化
{
  mciSendCommand(open.wDeviceID,MCI_CLOSE,0,0);
                                                //若初始化，则发送关闭命令到 MCI
}
open.lpstrElementName=str;                      //获取的路径传给 MCI 初始化结构体

open.lpstrDeviceType="mpegvideo";               //指定 MCI 的设备类型
err=mciSendCommand(0,MCI_OPEN,MCI_OPEN_TYPE|MCI_OPEN_ELEMENT|MCI_WAIT,(
DWORD)(LPVOID)&open);                           //初始化 MCI 设备
GetDlgItem(IDC_PLAY)->SetWindowText(text);
if(text=="播放")
{
```

```
if(err==0)                                    //若初始化成功
{
MCI PLAY PARMS play;
mciSendCommand(open.wDeviceID,MCI_PLAY,0,(DWORD)&play);
                                              //向 MCI 设备发送播放命令
GetDlgItem(IDC_PLAY)->SetWindowText("暂停");
}
else                                          //若设备初始化失败
{
mciGetErrorString(err,(LPSTR)str1,100);       //获取错误信息
    MessageBox(str1);                         //显示错误信息
    }
else
{
  mciSendCommand(open.wDeviceID,MCI_PAUSE,0,0);//向 MCI 设备发送暂停命令
  GetDlgItem(IDC_PLAY)->SetWindowText("播放");
}
```

用户在上面的程序中，看到函数 GetItemText()的第一个参数是 n。该参数表示用户在列表中选择的列表项索引值。用户可以在列表控件的单击消息响应函数 OnClickList1()中，获取当前所选择的项索引并将该索引值传递给参数 n。代码如下：

```
void CP2PView::OnClickList1(NMHDR* pNMHDR, LRESULT* pResult)
{
    int nItem;                                    //定义变量保存列表索引
    POSITION pos=m_list.GetFirstSelectedItemPosition();//获取用户单击的位置
if(pos!=NULL)                                     //判断单击位置是否为空
{
    nItem=m_list.GetNextSelectedItem(pos);        //获取单击位置的列表索引
    n=nItem;                                      //将该索引值赋给变量 n
}
    *pResult = 0;
}
```

用户通过以上两个步骤，已经实现了实例程序基本的 MP3 播放功能。下面将根据变量 n 的值对其他控制按钮的功能进行实现。

当用户需要停止当前所播放的 MP3 时，单击“停止”按钮可以实现该功能。并且当用户再次单击“播放”按钮时，程序所播放的 MP3 文件为当前文件的后面第三个 MP3 文件。代码如下：

```
void CP2PView::OnStop()
{
    mciSendCommand(open.wDeviceID,MCI_STOP,0,0);//向 MCI 设备发送停止命令
    if(n<m_list.GetItemCount())             //判断当前列表索引是否超出列表项目数
    {
        n+=3;                               //将索引值加 3
    }
    else                                    //如果当前索引值超出列表项目数
    {
        n=0;                                //将索引值设置为 0
    }
}
```

如果用户需要播放当前 MP3 文件的上一首或者下一首时，可以通过单击“上一首”或

者“下一首”按钮即可实现这些功能。首先，在“下一首”按钮的消息响应函数中添加功能代码。其代码如下：

```
void CP2PView::OnNext()                        //下一首按钮消息响应函数
{
    if(n<m_list.GetItemCount())                 //判断当前列表索引值是否大于列表总数
    {
        n+=1;
    CString str=m_list.GetItemText(n,3);//获取默认列表项中的 MP3 文件路径
if(open.wDeviceID)                              //判断指定 ID 的 MCI 设备是否初始化
{
  mciSendCommand(open.wDeviceID,MCI_CLOSE,0,0);
                                                //若初始化，则发送关闭命令到 MCI
}
open.lpstrElementName=str;                      //获取的路径传给 MCI 初始化结构体
open.lpstrDeviceType="mpegvideo";               //指定 MCI 的设备类型
err=mciSendCommand(0,MCI_OPEN,MCI_OPEN_TYPE|MCI_OPEN_ELEMENT|MCI_WAIT,(
DWORD)(LPVOID)&open);
}
    if(err==0)                                  //若初始化成功
{
MCI_PLAY_PARMS play;                            //则向 MCI 设备发送播放命令
mciSendCommand(open.wDeviceID,MCI_PLAY,0,(DWORD)&play);
    }
}
```

然后，在“上一首”按钮的消息响应函数中添加代码。其代码如下：

```
void CP2PView::OnPre()                              //上一首按钮消息响应函数
{
CString str;                                        //定义字符串变量
if(n==0)                                            //如果当前索引值为 0
{
    n=0;
}
else
{
n=n-1;                                              //否则，将当前索引值减一
}
str=m_list.GetItemText(n,3);                        //获取指定索引值项目的路径字符串
if(open.wDeviceID)                                  //判断指定 ID 的 MCI 设备是否存在
{
  mciSendCommand(open.wDeviceID,MCI_CLOSE,0,0);
                                                    //若存在，则发送关闭命令到 MCI 设备
}
open.lpstrElementName=str;                          //将获取到的路径字符串赋予该成员
open.lpstrDeviceType="mpegvideo";
err=mciSendCommand(0,MCI_OPEN,MCI_OPEN_TYPE|MCI_OPEN_ELEMENT|MCI_WAIT,(
DWORD)(LPVOID)&open);                               //向 MCI 设备发送初始化命令
if(err==0)                                          //若初始化 MCI 设备发生错误
{
MCI_PLAY_PARMS play;
mciSendCommand(open.wDeviceID,MCI_PLAY,0,(DWORD)&play);
                                                    //向 MCI 设备发送播放指令
}
}
```

10.3　客户机之间的连接

在本章实例中，主要向用户介绍使用 P2P 技术实现客户机之间的连接与数据传输等操作。因此，在本节中，将重点介绍客户机之间的连接编程。

10.3.1　创建套接字

与其他网络程序一样，P2P 网络播放器同样需要套接字的支持。所以，用户在实际编程时，其编程流程与套接字编程流程相同。在本节中，将向用户介绍编写 P2P 程序的一般流程以及方法，并且用户在套接字创建成功后，使用异步模式对该套接字进行操作的方法实现等。关于异步套接字的操作方法等已经在前面的章节中向用户进行详细的讲解。所以，在本节中，将不再向用户讲解这方面的基础知识。请用户复习前面的相关知识。

1．定义套接字对象

用户在编写任何网络程序时，都需要首先创建套接字对象或句柄以后，才能成功进行网络传输。因此，在本实例中，用户需要在视图类中定义套接字对象或句柄。由于本实例是基于 API 函数编写，所以在这里定义套接字句柄即可。代码如下：

```
class CP2PView : public CFormView
{
 public:
    ...                                             //省略部分变量定义
    SOCKET s1;                                      //定义套接字句柄变量
   }
```

在程序中，用户一共定义了两个套接字句柄，这是因为在 P2P 程序中需要计算机既可以是服务器又可以是客户端。其中套接字句柄 s1 是作为服务器套接字，而套接字句柄 s2 则作为客户端套接字。这样，用户所编写的 P2P 程序便具有了接收数据与主动搜索数据的功能了。

如果定义套接字句柄成功后，用户便可以在视图初始化函数 OnInitialUpdate()中，使用 API 函数 socket()创建套接字句柄。代码如下：

```
void CP2PView::OnInitialUpdate()
{
  WSADATA data;                                     //定义结构体变量
    DWORD ss=MAKEWORD(2,0);                         //定义套接字版本信息
   ::WSAStartup(ss,&data);                          //初始化套接字库
   s1=::socket(AF_INET,SOCK_STREAM,IPPROTO_TCP);
                                                    //创建套接字
   s2=::socket(AF INET,SOCK STREAM,IPPROTO TCP);
...                                                 //省略部分代码
   }
```

在程序中，用户使用 API 函数创建了两个套接字句柄 s1 和 s2。用户在后面的所有网络数据传输等操作，均是通过这两个套接字句柄进行实现。

2．绑定套接字句柄

如果用户成功创建套接字后，则需要将服务器套接字句柄与本地地址和端口号绑定在一起。这样，程序才能接收到其他客户机的数据请求。程序中，用户绑定服务器套接字可以使用 API 函数 bind()等实现。代码如下：

```
...                                                //省略部分代码
::gethostname((char*)&name,(int)sizeof(name)); //获得主机名字
hostent *p=::gethostbyname((char*)&name);      //将主机名转换为地址结构变量
in_addr *a=(in_addr*)*p->h_addr_list;          //获得本机 IP 地址
CString str=::inet_ntoa(a[0]);                 //将 IP 地址转换为主机 IP 地址
sockaddr_in addr;                              //定义套接字信息结构体变量
addr.sin_family=AF_INET;                       //填充该结构体中的成员变量
addr.sin_port=htons(80);
addr.sin_addr.S_un.S_addr=inet_addr(str);
::bind(s1,(sockaddr*)&addr,sizeof(addr));      //绑定套接字句柄到本地地址
::listen(s1,5);                                //监听套接字
...                                            //省略部分代码
```

在程序中，函数 listen()的第二个参数设置为 5，表示在服务器套接字 s1 上支持同时响应 5 个客户机的连接请求。

注意：如果用户对套接字编程的基础知识不熟悉，请复习前面相关章节的基础知识。

3．设置异步套接字

在本实例中，将采用异步套接字模式进行编程操作。其优点是当且仅当该异步套接字上有数据请求或传输时，程序才会调用相关的函数进行处理。代码如下：

```
...                                                    //省略部分代码
WSAAsyncSelect(s1,this->m_hWnd,WM_SOCK,FD_ACCEPT|FD_READ);//设置异步套接字
WSAAsyncSelect(s2,this->m_hWnd,WM_SOCK,FD_ACCEPT|FD_READ);
```

函数 WSAAsyncSelect 的作用是将指定的套接字句柄设置为异步模式，并指定该套接字上发生的网络事件以及处理这些事件的窗口消息等。用户需要自定义窗口消息 WM_SOCK，并且为该消息指定相应的消息响应函数。代码如下：

```
#define WM_SOCK WM_USER+100                        //自定义消息
class CP2PView : public CFormView
{
 protected:
    //{{AFX_MSG(CP2PView)
    ...                                            //省略部分代码
    afx_msg void Onsoc(WPARAM wParam,LPARAM lParam);//声明消息响应函数
}
```

然后，再在视图类的实现文件中添加套接字消息映射表。代码如下：

```
...                                                //省略部分代码
IMPLEMENT_DYNCREATE(CP2PView, CFormView)
BEGIN_MESSAGE_MAP(CP2PView, CFormView)
    //{{AFX_MSG_MAP(CP2PView)
```

```
    ...                                              //省略部分代码
    ON_BN_CLICKED(IDC_NEXT, OnNext)
    ON_BN_CLICKED(IDC_RADIO1, OnRadio1)
    ON_BN_CLICKED(IDC_RADIO2, OnRadio2)
    ON_NOTIFY(NM_DBLCLK, IDC_LIST1, OnDblclkList1)
    ON_NOTIFY(NM_CLICK, IDC_LIST1, OnClickList1)
    ON_MESSAGE(WM_SOCK,Onsoc)                        //套接字消息映射表
    //}}AFX_MSG_MAP
END_MESSAGE_MAP()
```

用户通过以上几个步骤，已经基本完成了异步套接字的设置。

10.3.2　使用 SOCKET 数组保存套接字句柄

因为基于 P2P 编写的程序可以响应多个客户机的连接请求，或者向多个客户机发送数据请求等特点。所以，在实例程序中，用户需要定义一个套接字数组保存相应的套接字句柄。

首先，用户在视图类中，定义一个 SOCKET 类型的套接字句柄数组。代码如下：

```
class CP2PView : public CFormView
{
  public:
    ...                                          //省略部分代码
    SOCKET s1,s2;                                //创建套接字句柄
    SOCKET s[5];                                 //创建套接字句柄数组
  }
```

在代码中，用户将套接字数组大小设置为 5。这是因为用户在使用监听函数 listen()监听套接字时，已经将其第二个参数设置成了 5。所以，用户在定义套接字数组时，需要将该数组的大小设置为 5。

为了避免程序在初始化时发生错误，用户需要在视图的初始化函数中对套接字句柄数组进行初始化。代码如下：

```
void CP2PView::OnInitialUpdate()
{
    CFormView::OnInitialUpdate();
   ...                                           //省略部分代码
   memset(&s,0,5);                               //将套接字数组初始化为 0
}
```

当监听套接字上有相应的网络事件到来时，程序便将可以在连接事件 FD_ACCEPT 中使用函数 accept()响应客户机的连接请求。在响应的同时，将服务器与客户机之间收发数据的套接字句柄存放于套接字句柄数组中。代码如下：

```
void CP2PView::Onsoc(WPARAM wParam,LPARAM lParam)
{
    switch (lParam)
    {
        case FD_ACCEPT:                          //连接事件
        if(i<5)                                  //判断套接字句柄数组的当前大小
            {
                s[i]=::accept(s1,NULL,NULL);
              WSAAsyncSelect(s[i],this->m_hWnd,WM_SOCK,FD_READ);
                                                 //将通信套接字也设置为异步模式
```

```
                ...                                     //省略部分代码
            }
...                                                     //省略部分代码
}
```

用户在编程时，按照上面的代码样式将服务器程序与发送连接请求的客户机的通信套接字保存在套接字数组中，然后调用函数 WSAAsyncSelect()将该通信套接字也设置为异步模式。这样，当实例程序需要搜索网络 MP3 文件时，便可以使用这些套接字向多个客户机发送搜索请求并等待数据传输。

10.4　传 输 数 据

用户创建好实例程序所需的套接字等变量以后，便可以通过网络进行数据传输了。首先，应该是某台客户机连接主计算机或另一台客户机，以获取该网络中的所有在线客户机的地址信息。然后，再使用客户端套接字向所在这些地址信息的计算机发出连接请求。如果连接成功，则继续请求数据资源。在本节中，将主要向用户介绍传输数据时所用的数据结构封装方法以及数据传输控制。

10.4.1　数据结构

在本章实例中，将继续使用第 9 章中所使用的 MP3 数据结构。但是，用户需要在原有数据的基础上加入 MP3 的实体数据成员，即 MP3 数据帧。修改后的 MP3 数据结构如下：

```
typedef struct mp3_str{                //自定义 MP3 标签帧结构体
char heade[3];                         //TAG 字符标记
char title[30];                        //音乐文件名称
char arti [30];                        //演唱者
char alb [30];                         //专辑
char year[4];                          //出版年份
char text[28];                         //备注内容
} mp3struct;
typedef struct mp3_s{                  //自定义 MP3 结构体（修改后的 MP3 数据结构）
char *data;                            //数据帧指针
mp3struct mp3b;                        //标签帧结构体变量
}mp3;
```

用户在修改后的 MP3 数据结构中可以看到，新添加的数据成员是字符指针 data，并将其定义为数据帧指针。本书使用该数据结构操作 MP3 文件的方法是当用户使用该数据结构读取 MP3 文件时，需要使用 MP3 文件的总大小剪掉标签帧的大小即 128 字节后，再将数据帧指针指向数据帧的首地址并将其读出。而写入 MP3 文件时，应该首先按照数据帧的大小将数据帧写入 MP3 文件后，再将标签帧中的数据继续写入 MP3 文件即可。例如，用户使用该结构读取并发送 MP3 文件，代码如下：

```
...                                              //省略部分代码
mp3 mp3data;                                     //定义 MP3 数据结构变量
CString str;
```

```
GetDlgItem(IDC_EDIT1)->GetWindowText(str); //获取文件路径
CFilefile(str,
CFile::modeRead|CFile::modeCreate|CFile::modeNoTruncate|CFile::typeBina
ry);
                                            //定义文件对象并关联 MP3 文件
        int n=file.GetLength()-128;         //获得数据帧的大小
        char data1[n]={0};                  //定义数据帧缓冲区
        mp3data.data=&data1;                //将数据首地址赋予 MP3 数据帧成员
        file.Read(mp3data.data,n);          //读取数据帧中的数据到缓冲区
        file.Read(mp3data.mp3b,128);        //读取 MP3 标签帧中的数据
        send(s1, mp3data.data,n,0);         //发送 MP3 数据帧数据
        send(s1, mp3data.mp3b,n,0);         //发送 MP3 标签帧数据
...                                         //省略部分代码
```

用户在以上代码中，主要将指定 MP3 文件的数据帧和标签帧从文件中读取到指定缓冲区中保存。如果用户接收到 MP3 文件，需要将该文件写入本地磁盘中，代码如下：

```
...                                         //省略部分代码
mp3 mp3data;                                //定义 MP3 结构体对象
CString str;
GetDlgItem(IDC_EDIT1)->GetWindowText(str); //获取文件路径
CFilefile(str,
CFile::modeRead|CFile::modeCreate|CFile::modeNoTruncate|CFile::typeBina
ry);
                                            //定义文件对象并关联 MP3 文件
        recv(s,(char *)n,1,0);              //接收数据帧大小值
        char data1[n]={0};                  //定义数据帧缓冲区
        recv(s1,&data1,n,0);                //接收数据帧数据
recv(s1,& mp3data.mp3b,128,0);              //接收标签帧数据
        mp3data.data=&data1;                //将数据首地址赋予 MP3 数据帧成员
        file.Write(mp3data.data,n);         //写入数据帧中的数据到缓冲区
          file.Write(mp3data.mp3b,128);     //写入 MP3 标签帧中的数据
...                                         //省略部分代码
```

在代码中，用户首先调用函数接收 MP3 文件的数据帧与标签帧数据到指定缓冲区中。然后，用户再将这些数据写入文件中保存。

10.4.2　数据传输控制

数据传输控制是指用户实际编程时，需要控制数据传输的顺序等。该实例中，将查询 MP3 文件的命令以字符 c 代替，表示请求查询 MP3 资源。当接收方接收到该命令后，将在本地计算机中查找相应的文件并读取到缓冲区中进行发送，而发送命令方则接收数据并写入文件。

1. 发送查询命令

在实例中，将定义字符 c 为查询命令字符。所以，当用户编写程序时，使用客户端套接字将该字符发送到对方计算机即可。代码如下：

```
...                                             //省略部分代码
char ml='c';                                    //定义命令字符
send(s2,ml,1,0);                                //发送查询命令字符
```

```
...                                                          //省略部分代码
```

因此，在实例程序中，用户应该将发送命令字符的功能放在“查找网络文件”按钮的消息响应函数中。代码如下：

```
void CP2PView::OnFind()
{
char ml='c';                                        //定义命令字符
socketaddr in addr;                                 //定义地址结构
addr.sin_family=AF_INET;                            //填充地址结构中的成员
addr.sin port=htons(80);
addr.sin addr.S un.S addr=inet addr(str);
if(connect(s2,( sockaddr*)addr,sizeof(addr))!=-1)
                                                    //连接远程计算机
{
send(s2,ml,1,0);                                    //发送查询命令字符
}
}
```

在代码中，变量 str 表示远程计算机的 IP 地址。用户在使用实例程序时，需要提供一个远程计算机的 IP 地址才能使用。但是，用户只需输入一个远程计算机的 IP 地址，当程序连接后会自动获取其他计算机的 IP 地址。

2．接收命令并发送文件

当接收方客户机接收到查询命令之后，会在本地计算机中查询 F 盘中的所有 MP3 文件，并将这些文件读取到缓冲区中等待发送。代码如下：

```
void CP2PView::Onsoc(WPARAM wParam,LPARAM lParam)
{
    switch (lParam)
{
     ...                                            //省略部分代码
case FD_READ:                                       //读取事件
char mingl='';                                      //定义字符准备接收命令
recv(s2,mingl'1'0);                                 //接收命令字符
if(mingl && 'c')                                    //判断命令字符
{
CString strTmp=" F:\\音乐\\";                        //假设指定目录
strcat(strTmp.GetBuffer(0),"\\*.mp3");              //连接后缀名
findfile.FindFile(strTmp);                          //查找文件
while(findfile.FindNextFile())
{
str2=findfile.GetFilePath();                        //获取文件路径
CFile file(str2,CFile::modeRead);                   //创建文件对象
int n=file.GetLength()-128;                         //获得数据帧的大小
CString filetitle=file.GetFileTitle();              //获取文件标题
        char data1[n]={0};                          //定义数据帧缓冲区
        mp3data.data=&data1;                        //将数据首地址赋予 MP3 数据帧成员
        file.Read(mp3data.data,n);                  //读取数据帧中的数据到缓冲区
        file.Read(mp3data.mp3b,128);                //读取 MP3 标签帧中的数据
        send(s1, n,1,0);                            //发送文件数据帧大小
```

```
        send(s1, filetitle,sizeof(filetitle),0);//发送文件名
        send(s1, mp3data.data,n,0);              //发送 MP3 数据帧数据
        send(s1, mp3data.mp3b,n,0);              //发送 MP3 标签帧数据
...                                              //省略部分代码
}
}
```

在代码中，用户使用循环结构读取每一个搜索的 MP3 文件，并将相关的数据发送到请求方计算机。

3．接收文件数据并创建文件

当请求方计算机接收到数据时，程序会首先判断用户是否希望保存接收到的文件。若用户选择保存，则应该按照一定的顺序将接收到的数据写入文件中。否则，文件数据将被存放于临时文件中。主要代码如下：

```
void CP2PView::Onsoc(WPARAM wParam,LPARAM lParam)
{
    switch (lParam)
{
     ...                                     //省略部分代码
case FD_READ:                                //读取事件
mp3 mp3data;                                 //定义 MP3 结构体对象
CString str="F:\音乐\";                      //定义目录字符串
CString name;                                //定义命令字符串
recv(s1,name.GetBuffer(0),sizeof(name),0); //接收文件标题
strcat(strcat(str.GetBuffer(0),name);       //连接后缀名
CFilefile(str,
CFile::modeRead|CFile::modeCreate|CFile::modeNoTruncate|CFile::typeBina
ry);
                                             //定义文件对象并关联 MP3 文件
        recv(s,(char *)n,1,0);               //接收数据帧大小值
        char data1[n]={0};                   //定义数据帧缓冲区
        recv(s1,&data1,n,0);                 //接收数据帧数据
recv(s1,& mp3data.mp3b,128,0);               //接收标签帧数据
        mp3data.data=&data1;                 //将数据首地址赋予 MP3 数据帧成员
        file.Write(mp3data.data,n);          //写入数据帧中的数据到缓冲区
          file.Write(mp3data.mp3b,128);      //写入 MP3 标签帧中的数据
...                                          //省略部分代码
```

在代码中，用户首先接收文件名并在指定目录中创建 MP3 文件。然后，按照顺序接收 MP3 文件的数据帧以及标签帧并将其写入文件中。当用户将所有数据写入文件后，需要将该文件的相关信息添加到播放列表中。代码如下：

```
...                                              //省略部分代码
int nRow=m list.InsertItem(m list.GetItemCount()+1, mp3data.mp3b.title);
                                                 //插入行
m_list.SetItemText(nRow,1, mp3data.mp3b.arti); //设置数据
if(mp3data.mp3b.heade && "TAG")                  //判断是否为 MP3 文件
{
    CString mp3="MP3";                           //定义并初始化字符串
m_list.SetItemText(nRow,2,mp3);                  //设置数据
}
m_list.SetItemText(nRow,3,file.GetFilePath()); //添加文件路径
```

注意： 用户编程时，在接收文件数据或将其写入文件时，都应该严格按照对方读取文件数据和发送文件数据时的顺序进行。否则，数据顺序将发生错乱。

10.5　使用多线程进行数据传输与播放

由于程序在发送查询命令后，会等待另一方计算机的数据。而在这一段时间中，程序将一直处于等待状态，如果此时程序没有任何反应，那么用户会认为程序死锁了。为了避免这种情况的发生，用户在编写程序时，可以利用多线程编程来解决这个问题。

10.5.1　发送线程

用户在实例程序中，需要创建一个线程用于发送数据。首先，用户需要在视图头文件中定义线程函数 Senddata()，代码如下：

```
...                                             //省略部分代码
DWORD WINAPI Senddata(LPVOID lpParameter);      //声明线程函数
class CP2PView : public CFormView
{
...                                             //省略部分代码
  }
```

在代码中，声明线程函数 Senddata()时，必须将其声明在类外。否则，线程函数将调用失败。然后，用户需要实现该线程函数的功能。实现代码如下：

```
DWORD WINAPI Senddata(LPVOID lpParameter)
{
    CString strTmp=" F:\\音乐\\";              //假设指定目录
strcat(strTmp.GetBuffer(0),"\\*.mp3");      //连接后缀名
findfile.FindFile(strTmp);                  //查找文件
while(findfile.FindNextFile())              //查找下一个文件
{
str2=findfile.GetFilePath();                //获取文件路径
CFile file(str2,CFile::modeRead);           //创建文件对象
int n=file.GetLength()-128;                 //获得数据帧的大小
CString filetitle=file.GetFileTitle();      //获取文件标题
        char data1[n]={0};                  //定义数据帧缓冲区
        mp3data.data=&data1;                //将数据首地址赋予 MP3 数据帧成员
        file.Read(mp3data.data,n);          //读取数据帧中的数据到缓冲区
    file.Read(mp3data.mp3b,128);            //读取 MP3 标签帧中的数据
        send(s1, n,1,0);                    //发送文件数据帧大小
        send(s1, filetitle,sizeof(filetitle),0);//发送文件名
        send(s1, mp3data.data,n,0);         //发送 MP3 数据帧数据
        send(s1, mp3data.mp3b,n,0);         //发送 MP3 标签帧数据
    return 1;
}
```

最后，用户需要在套接字的读取事件 FD_READ 中，使用函数 CreateThread()创建发送线程发送文件数据即可。代码如下：

```
void CP2PView::Onsoc(WPARAM wParam,LPARAM lParam)
{
    switch (lParam)
{
  ...                                           //省略部分代码
case FD READ:
      char mingl='';                            //定义字符准备接收命令
recv(s2,mingl,1,0);                             //接收命令字符
if(mingl && 'c')                                //判断命令字符
{
    ::CreateThread(NULL,0, Senddata,NULL,0,NULL);  //创建线程
    break;
}
}
```

这样，用户便通过函数 CreateThread()创建了发送线程，当线程函数 Senddata()启动时，首先接收对方发送的命令字符并判断。如果是正确的命令字符，则启动线程函数搜索文件并发送文件。否则，程序将准备接收对方发送的文件数据并创建文件保存数据。在 10.5.2 节中，将向用户介绍用多线程程序对数据进行接收保存。

10.5.2　接收线程

用户在实例程序中，使用接收线程接收数据将节省系统资源。接收线程也需要声明接收线程函数和实现接收线程函数，最后使用函数 CreateThread()启动该线程。其方法与发送线程的方法相同，所以在这里本节将不再赘述，请用户参考 10.5.1 节的内容。在接收线程函数中，主要根据用户的选择决定是否保存接收到的文件数据。代码如下：

```
DWORD WINAPI Recvdata(LPVOID lpParameter)
{
mp3 mp3data;                                    //定义 MP3 结构体对象
CString str="F:\音乐\";                         //定义目录字符串
CString name;                                   //定义命令字符串
recv(s1,name.GetBuffer(0),sizeof(name),0); //接收文件标题
strcat(strcat(str.GetBuffer(0),name);           //连接后缀名
CFilefile(str,
CFile::modeRead|CFile::modeCreate|CFile::modeNoTruncate|CFile::typeBina
ry);
                                                //定义文件对象并关联 MP3 文件
        recv(s,(char *)n,1,0);                  //接收数据帧大小值
        char data1[n]={0};                      //定义数据帧缓冲区
        recv(s1,&data1,n,0);                    //接收数据帧数据
recv(s1,& mp3data.mp3b,128,0);                  //接收标签帧数据
        mp3data.data=&data1;                    //将数据首地址赋予 MP3 数据帧成员
       if(GetDlgItem(IDC_RADIO1)->GetCheck())//判断用户是否选择保存文件
      {
        file.Write(mp3data.data,n);                 //写入数据帧中的数据到缓冲区
          file.Write(mp3data.mp3b,128);             //写入 MP3 标签帧中的数据
          }
int nRow=m_list.InsertItem(m_list.GetItemCount()+1, mp3data.mp3b.title);
                                                    //插入行
m_list.SetItemText(nRow,1, mp3data.mp3b.arti); //设置数据
if(mp3data.mp3b.heade && "TAG")                     //判断是否为 MP3 文件
{
```

```
    CString mp3="MP3";                          //定义并初始化字符串
m_list.SetItemText(nRow,2,mp3);                 //设置数据
}
m_list.SetItemText(nRow,3,file.GetFilePath()); //添加文件路径
        file.Close();                           //关闭文件
}
```

然后，用户便可以在套接字的读取事件中，调用函数 CreateThread()启动接收线程了。代码如下：

```
void CP2PView::Onsoc(WPARAM wParam,LPARAM lParam)
{
    switch (lParam)
{
  ...                                           //省略部分代码
case FD_READ:
      char mingl='';                            //定义字符准备接收命令
recv(s2,mingl,1,0);                             //接收命令字符
if(mingl && 'c')                                //判断命令字符
{
    ...                                         //省略发送线程的部分代码
}
else
{
  ::CreateThread(NULL,0, Recvdata,NULL,0,NULL);//创建接收线程
    break;
}
}
```

到这里为止，用户已经实现了全部的代码编写。保存并编译运行程序，用户在界面中单击“查找网络文件”按钮，程序将发送查询命令字符。如果客户机查询成功，则返回文件数据，如图 10.24 所示。否则，程序将向连接到该台客户机的所有 P2P 客户机，发送查询命令并使用异步套接字消息响应函数等待接收数据。

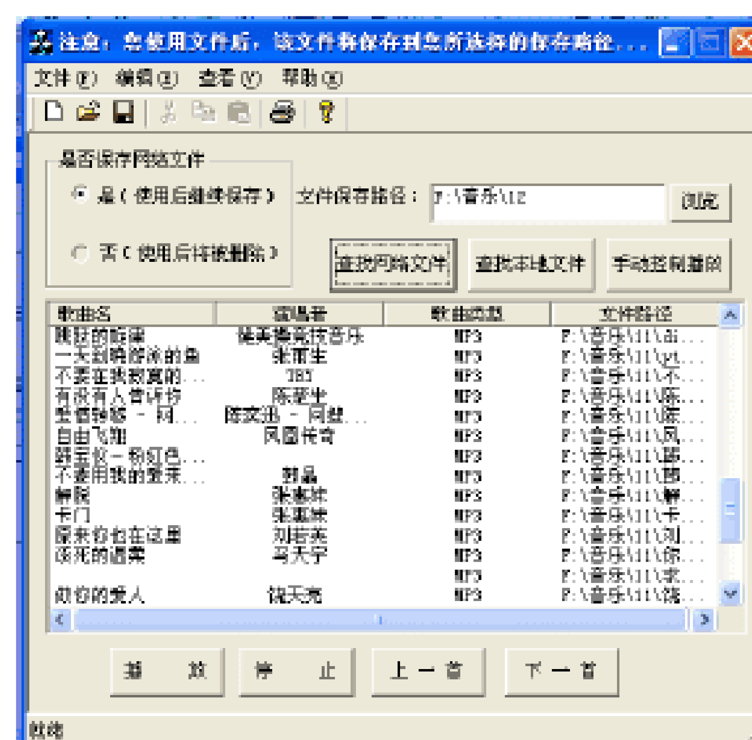


图 10.24　使用网络查找 MP3 文件

如图 10.24 所示，用户选择了保存搜索到的网络 MP3 文件，并且将这些文件保存在路径为“F:\音乐\12”的目录中。同时，用户也可使用实例程序播放这些通过网络接收的文件。请用户参照书本中所讲的理论知识并结合随书光盘中的实例代码进行学习。

10.6　小　　结

通过本章学习，用户应该掌握基于 P2P 所编写程序的基础知识。本章主要向用户讲述了在 VC 中怎样实现 P2P 程序、搜索网络 MP3 文件以及本地 MP3 文件的相关方法。

注意： 用户在调试随书光盘中实例程序时，需要运行多个实例程序，并且需要手动将代码中的服务器套接字和客户端套接字地址结构中的端口成员修改为相同。否则，程序之间将不能实现通信。

第 11 章　Q 版聊天软件

用户在日常生活中，最常用的即时通信软件应该是 QQ 聊天软件了。QQ 软件以其独特的功能与丰富的程序界面获得了很多用户的青睐。所以，在本章中将向用户介绍怎样在 VC 环境下，制作仿 QQ 实例程序界面的原理及其方法。

11.1　界 面 设 计

在引言中向用户提到了 QQ 之所以受到很多用户的青睐，最大的原因在于其丰富的程序界面。所以，在本节中，将向用户主要介绍服务器端和客户端的界面程序设计方法。特别是实例程序的窗口界面等需要与 QQ 真实界面大致相同，需要用户具有十分娴熟的界面设计功底。

11.1.1　服务器端

在实例程序的服务器端，主要功能是显示客户端的连接情况和接收到的数据等信息。因此，在 VC 中，使用 MFC 应用程序向导创建基于对话框类型的实例工程，并且在对话框面板中添加相应的子控件，以丰富界面的功能。

1．创建实例工程

用户在 VC 中，创建 Q 版实例工程。其步骤如下：

（1）选择“文件”|“新建”命令，打开“新建”对话框，如图 11.1 所示。

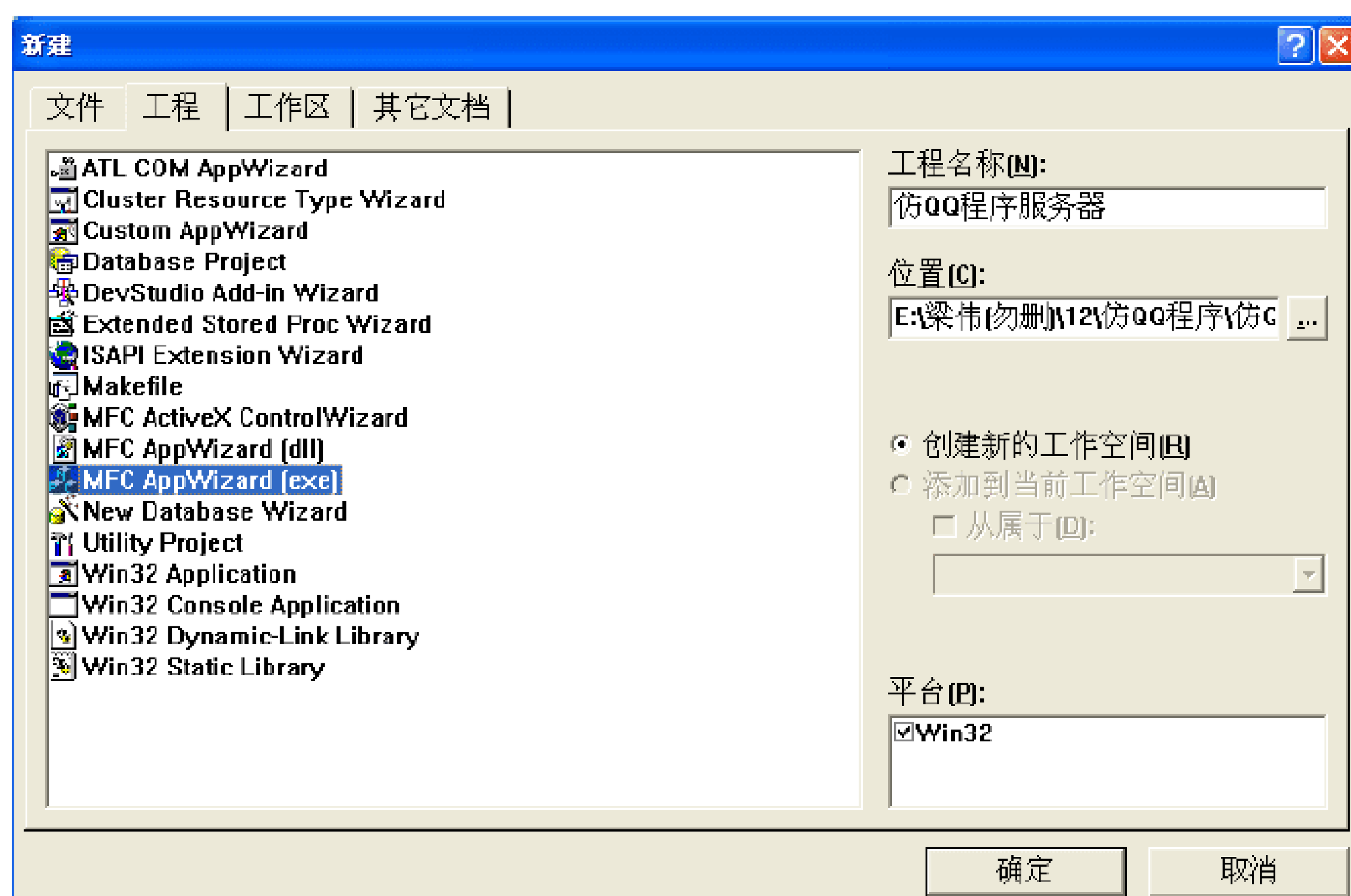


图 11.1　新建工程对话框

用户在新建工程对话框中，可以在“工程”选项卡中，设置新建工程的类型为 MFC AppWizard[exe]。然后，在相应的编辑框中，修改工程名为“仿 QQ 程序服务器”以及工程路径。

（2）单击“确定”按钮，选择应用程序类型为“基本对话框”。单击“下一步”按钮，选择应用程序的相关设置信息，如图 11.2 所示。

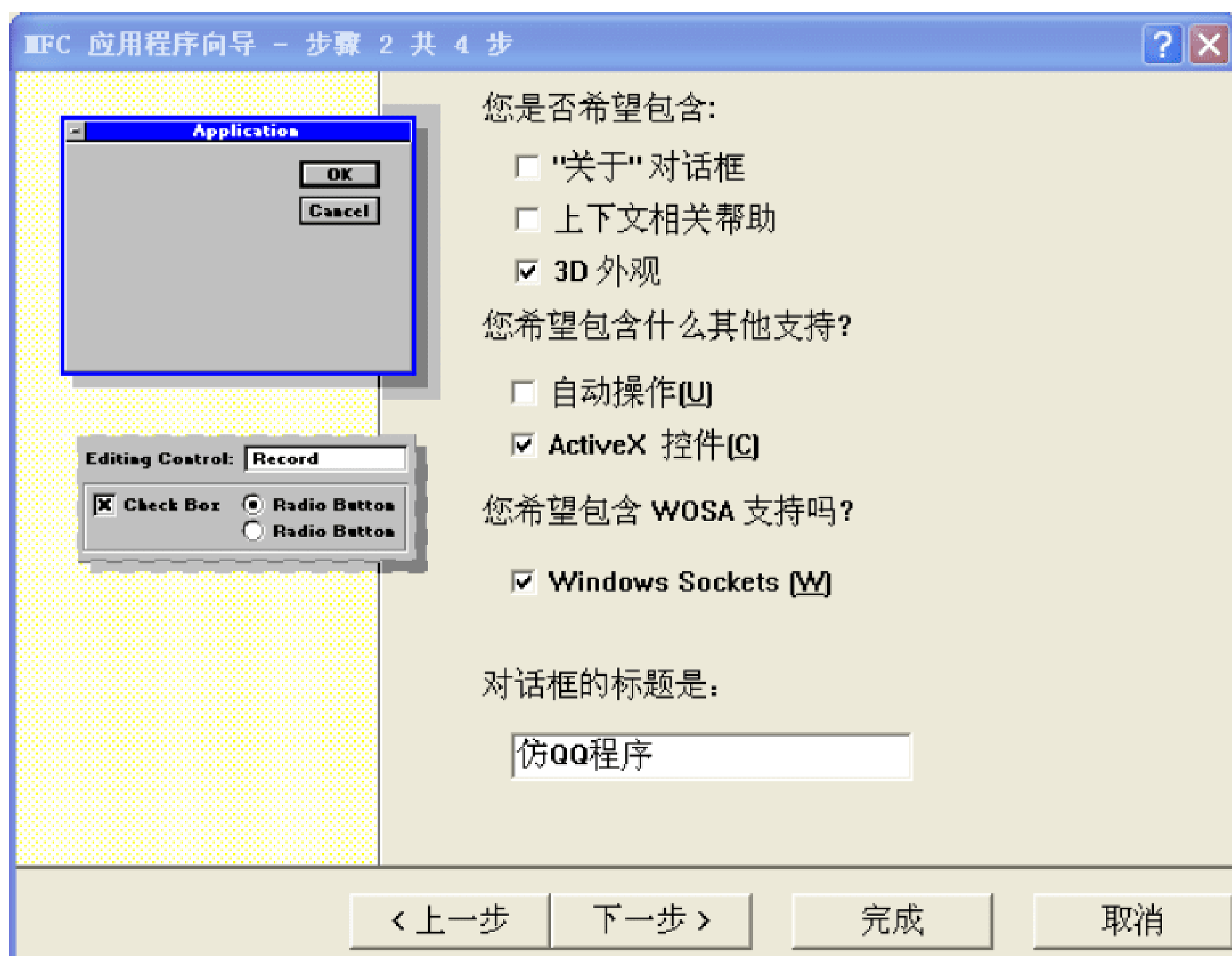


图 11.2　设置应用程序的相关信息

用户在应用程序设置中，必须选择 Windows Sockets 复选框。因为该实例程序需要网络套接字的相关库的支持，才能通过网络进行通信。否则，用户只能在编写代码时，手动添加相关代码。单击“下一步”按钮，选择生成源文件备注，如图 11.3 所示。

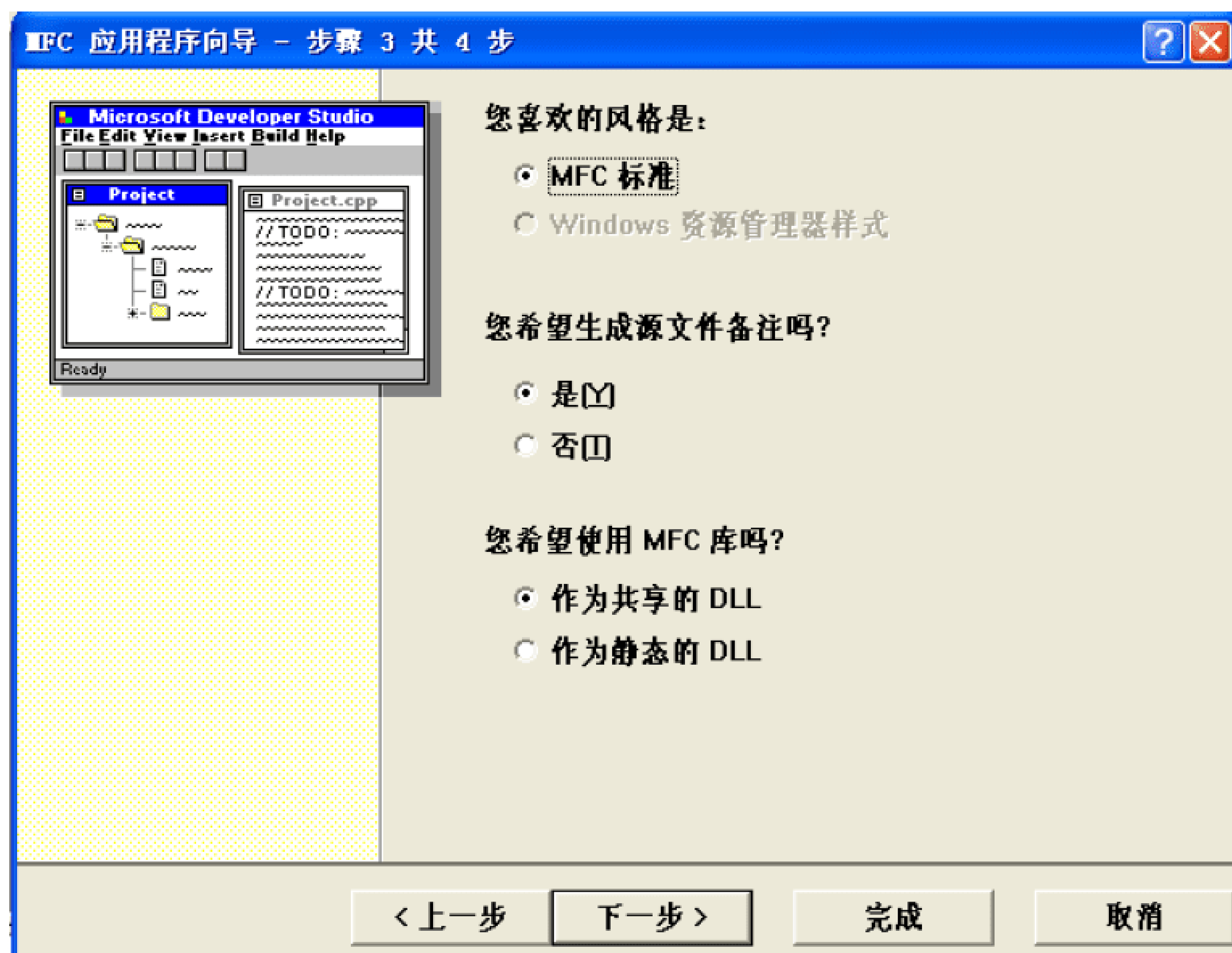


图 11.3　生成源文件备注

设置完毕以后，用户可以直接单击“完成”按钮，完成服务器实例工程的创建。

2．服务器界面设计

首先，用户在对话框面板中，使用鼠标拖入一个列表控件并将其拖拉到合适大小。该列表控件主要用于显示当前连接到服务器的用户昵称、IP 地址等相关信息，如图 11.4 所示。

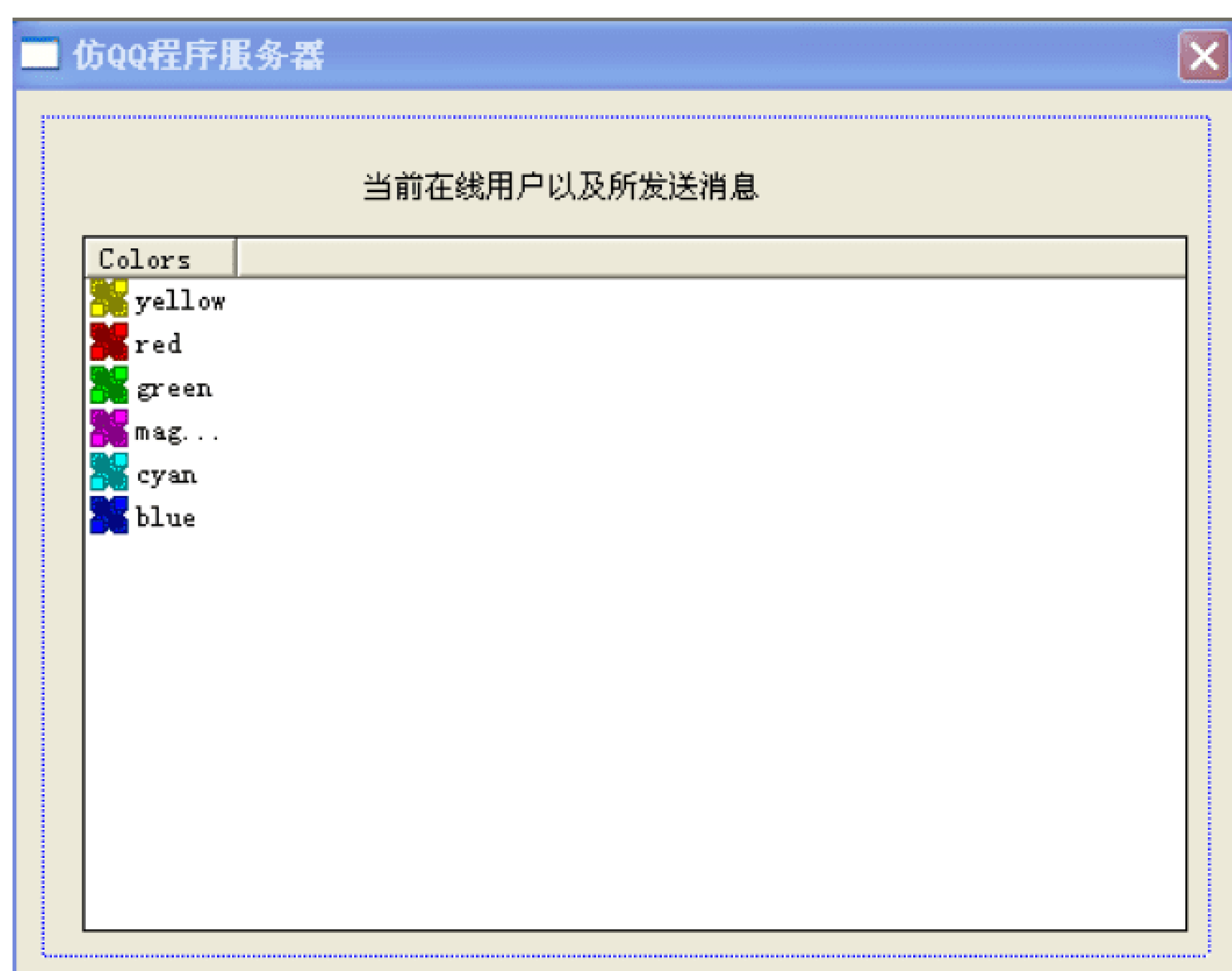


图 11.4　调整后的实例界面

但是，用户在真正使用该实例程序时，常常需要在服务器端向指定的客户端发送消息。因此，用户还需要在工程中添加一个消息发送对话框。该对话框是在用户发送消息时弹出。

在 VC 主界面中，选择“插入”|“资源”命令，即可弹出插入资源对话框，如图 11.5 所示。



图 11.5　插入资源对话框

用户在该对话框中，选择资源类型为 Dialog。然后单击“新建”按钮，程序将在实例工程中新建一个对话框。消息对话框添加成功后，用户可以编辑该对话框界面以达到实用效果，如图 11.6 所示。

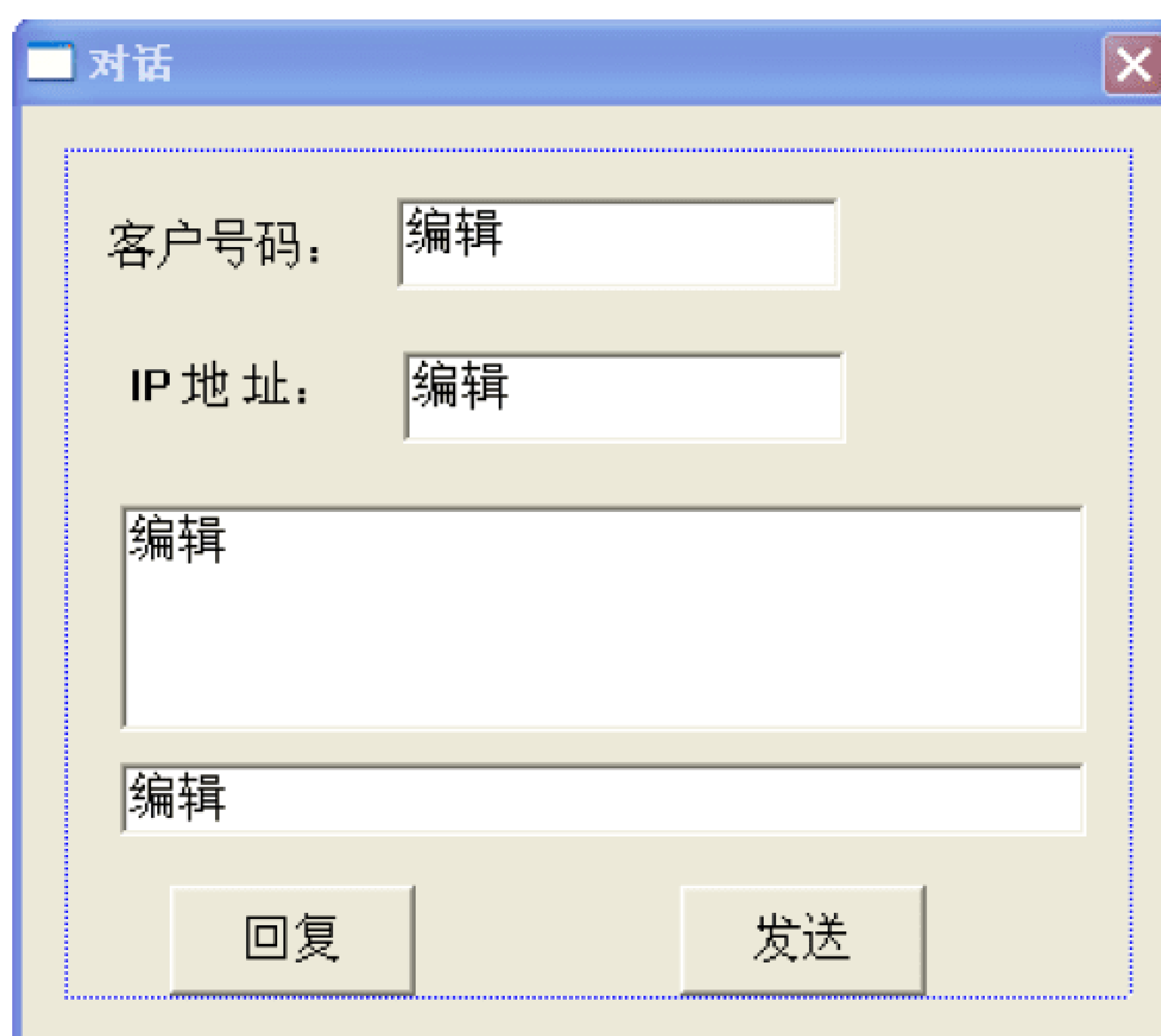


图 11.6　消息对话框界面

在消息对话框界面中，可以向用户显示连接客户端的号码、IP 地址等信息。其中控件 ID 以及功能等如表 11.1 所示。

表 11.1　消息对话框界面中各控件的ID以及作用

控　件　ID	控 件 类 型	控 件 作 用
IDC_NUM	编辑框	显示客户端号码
IDC_IP	编辑框	显示客户端IP
IDC_TEXT	编辑框	显示客户端所发送的信息
IDC_TEXT2	编辑框	发送到客户端的信息
IDC_RELAY	按钮	用于回复客户端消息
IDC_SEND	按钮	发送信息

3．为对话框资源关联新类

用户在实例工程中，使用新建对话框前，必须为其关联一个新类。否则，用户将不能使用该对话框。首先，在 VC 环境下，使用快捷键 Ctrl+W 弹出 MFC 向导对话框的同时将弹出 Adding a Class 对话框，如图 11.7 所示。用户选择第一个选项，表示创建一个新类。然后单击 OK 按钮，将弹出 New Class 对话框，如图 11.8 所示。

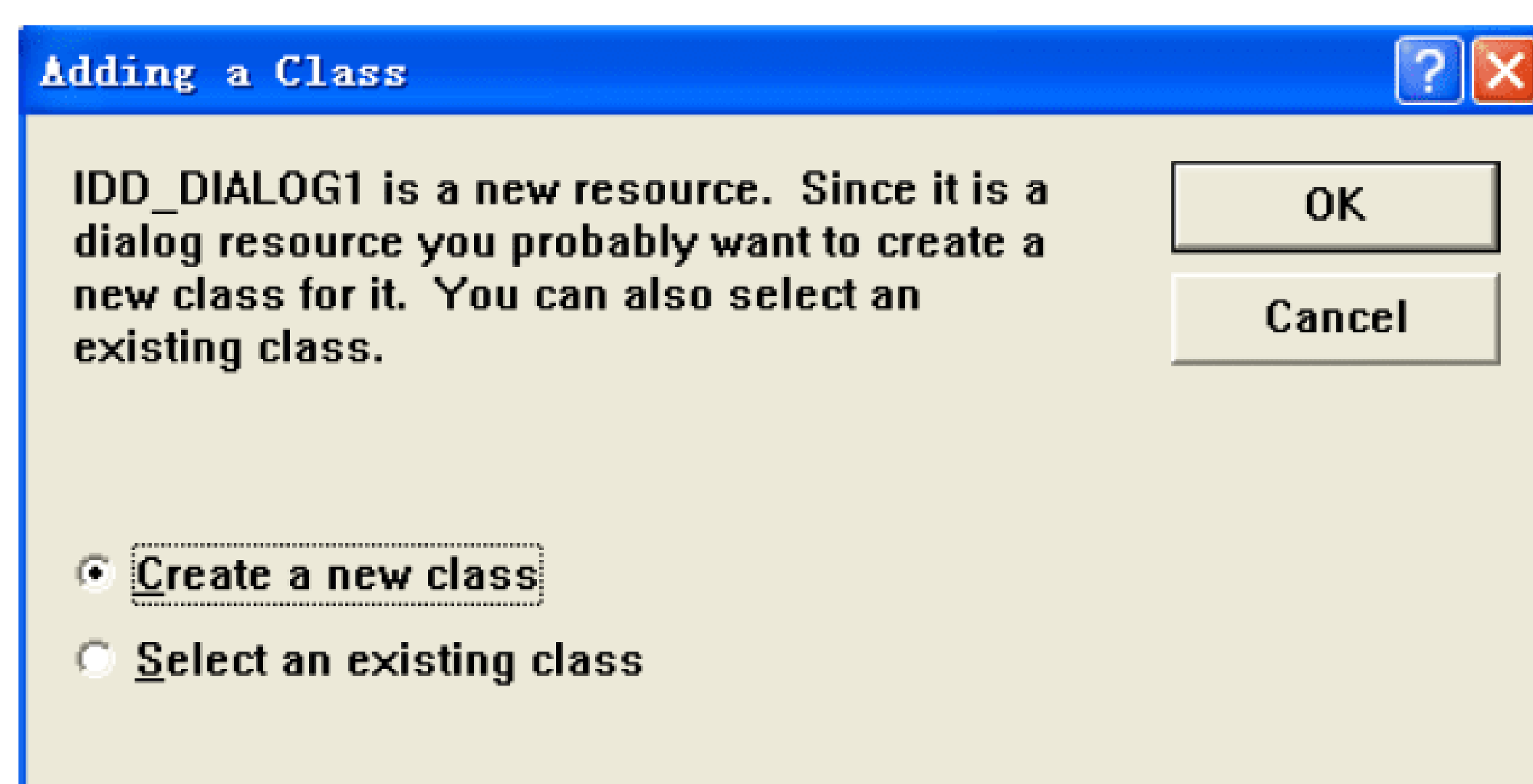


图 11.7　Adding a Class 对话框

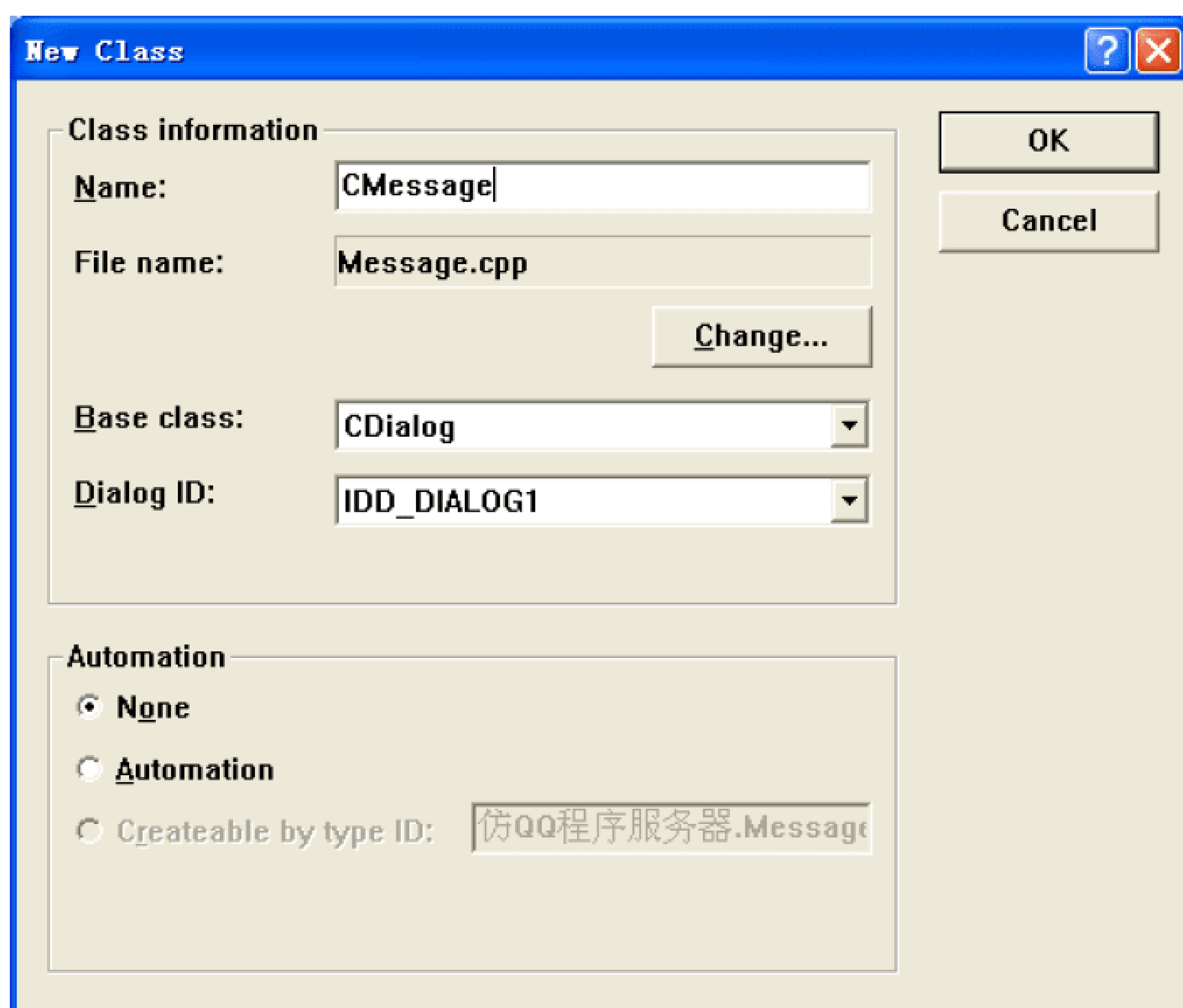


图 11.8　New Class 对话框

在该对话框中，用户可以修改新类名及其基类名等。在本章中，将新类名指定为 CMessage，将其基类名指定为 CDialog。然后，单击 OK 按钮完成新类的添加。

4．服务器界面初始化

在服务器端，界面初始化包括了主对话框以及消息显示对话框界面的初始化。下面将分别向用户讲解程序界面的初始化。

首先，当服务器程序启动时，程序应该将列表控件的各个标题显示出来。所以，用户在实例窗口的初始化函数 OnInitDialog()中，应该使用列表控件类的相关函数对标题进行设置。代码如下：

```
BOOL CQQDlg::OnInitDialog()
{
    CDialog::OnInitDialog();
   ...                                       //省略部分代码
LVCOLUMN lv;                                 //定义列表结构体变量
lv.mask=LVCF_TEXT|LVCF_FMT|LVCF_WIDTH;       //初始化结构体各个成员
lv.fmt=LVCFMT_CENTER;
lv.pszText="用户号码";                        //设置列表标题
lv.cx=120;                                   //指定该列宽度
m_list.InsertColumn(0,&lv);                  //插入指定列
lv.pszText="IP 地址";                         //修改列标题
m_list.InsertColumn(1,&lv);                  //插入指定列
lv.pszText="用户类型";                        //修改列标题
m_list.InsertColumn(2,&lv);                  //插入指定列
}
```

用户已经完成了对主对话框界面的初始化工作了，保存并编译运行程序，如图 11.9 所示。然后，消息对话框的初始化工作主要为界面中的控件状态及其显示内容的初始化。其中，控件状态的显示效果如图 11.10 所示。

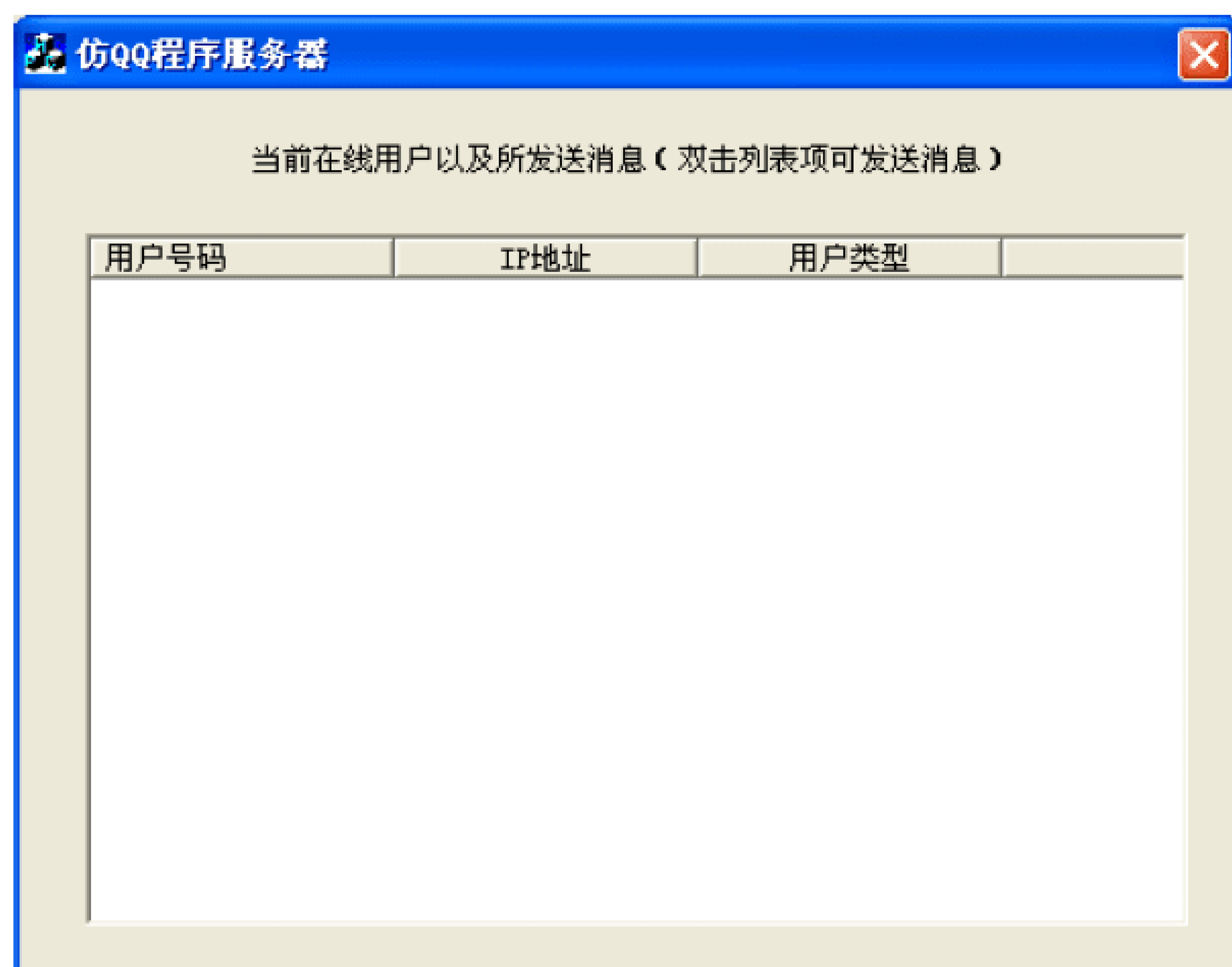


图 11.9　主对话框界面初始化

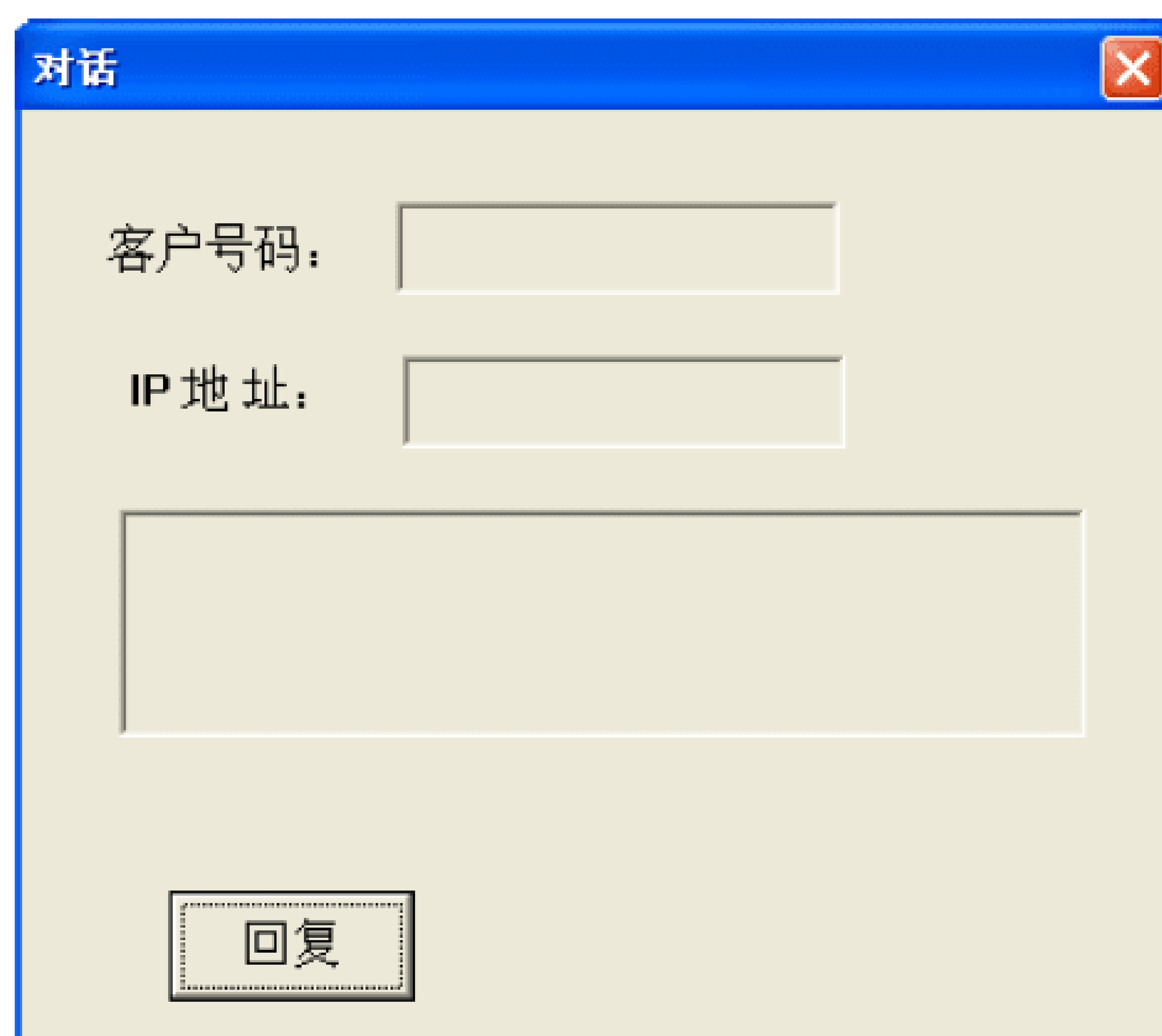


图 11.10　消息对话框界面初始化

用户在程序中，实现消息对话框初始化的代码如下：

```
BOOL CMessage::OnInitDialog()                          //消息对话框初始化函数
{
    CDialog::OnInitDialog();
    GetDlgItem(IDC_NUM)->EnableWindow(false);          //禁用客户端号码编辑框
    GetDlgItem(IDC_IP)->EnableWindow(false);           //禁用 IP 地址编辑框
    GetDlgItem(IDC_TEXT)->EnableWindow(false);         //禁用消息显示框
    GetDlgItem(IDC_TEXT2)->ShowWindow(false);          //隐藏消息发送框
    GetDlgItem(IDC_SEND)->ShowWindow(false);           //隐藏发送按钮
    ...                                                //省略部分代码
    return TRUE;
}
```

接下来，用户需要在消息对话框类 CMessage 中为各个子控件添加相应的变量，以实现当程序初始化时，便于设置各个控件的显示内容。在 VC 中添加成员变量的方法是使用快捷键 Ctrl+W，打开应用程序向导对话框的选项卡 Member Variables，如图 11.11 所示。

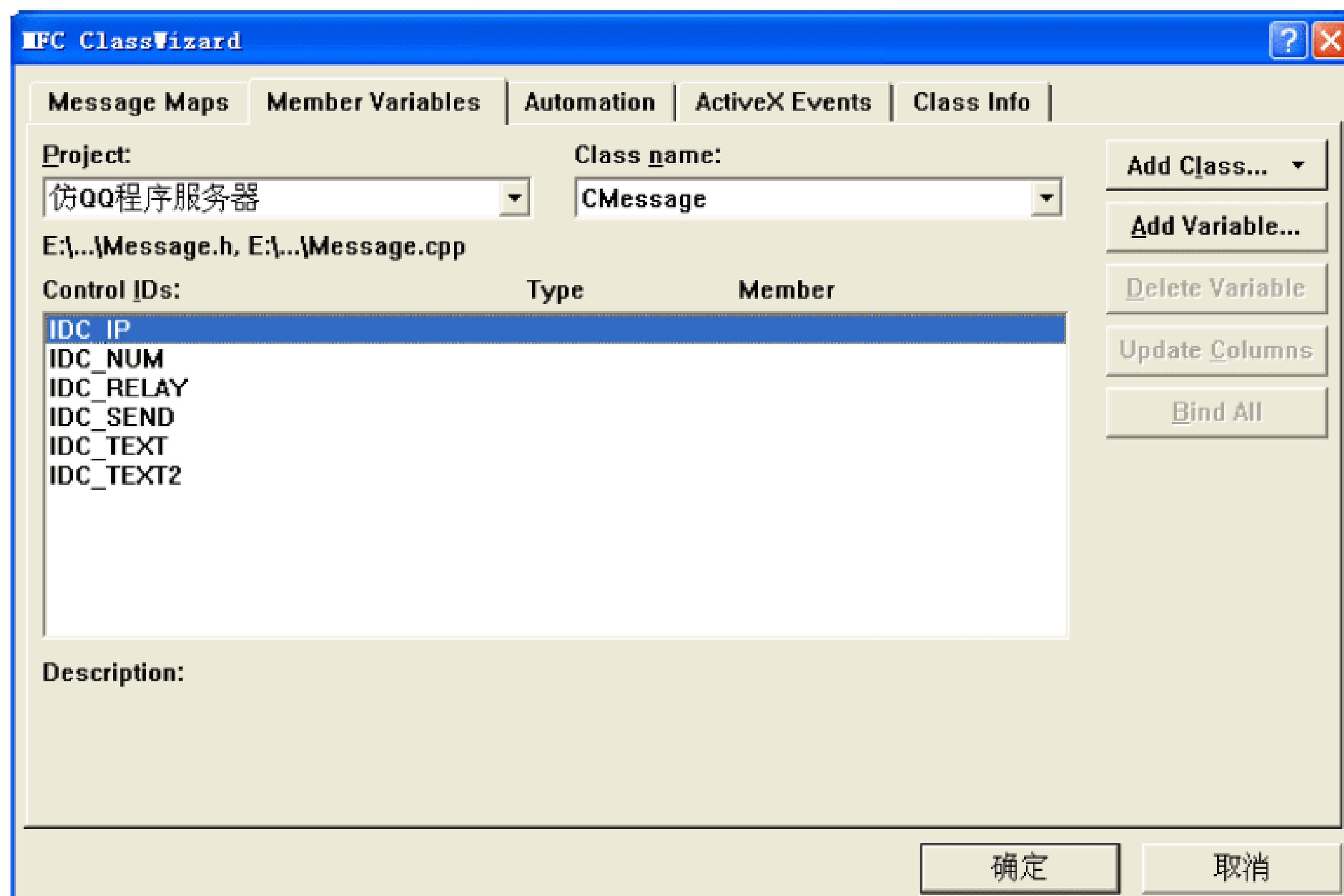


图 11.11　添加成员变量属性页

用户在该属性页中，可以为消息对话框中的子控件关联相应的控件变量。例如，为控件 IDC_IP 添加相应控件变量的方法是，在 Control IDs 列表中选择 IDC_IP，然后单击 Add Variable 按钮。此时，程序将弹出 Add Member Variable 对话框，如图 11.12 所示。

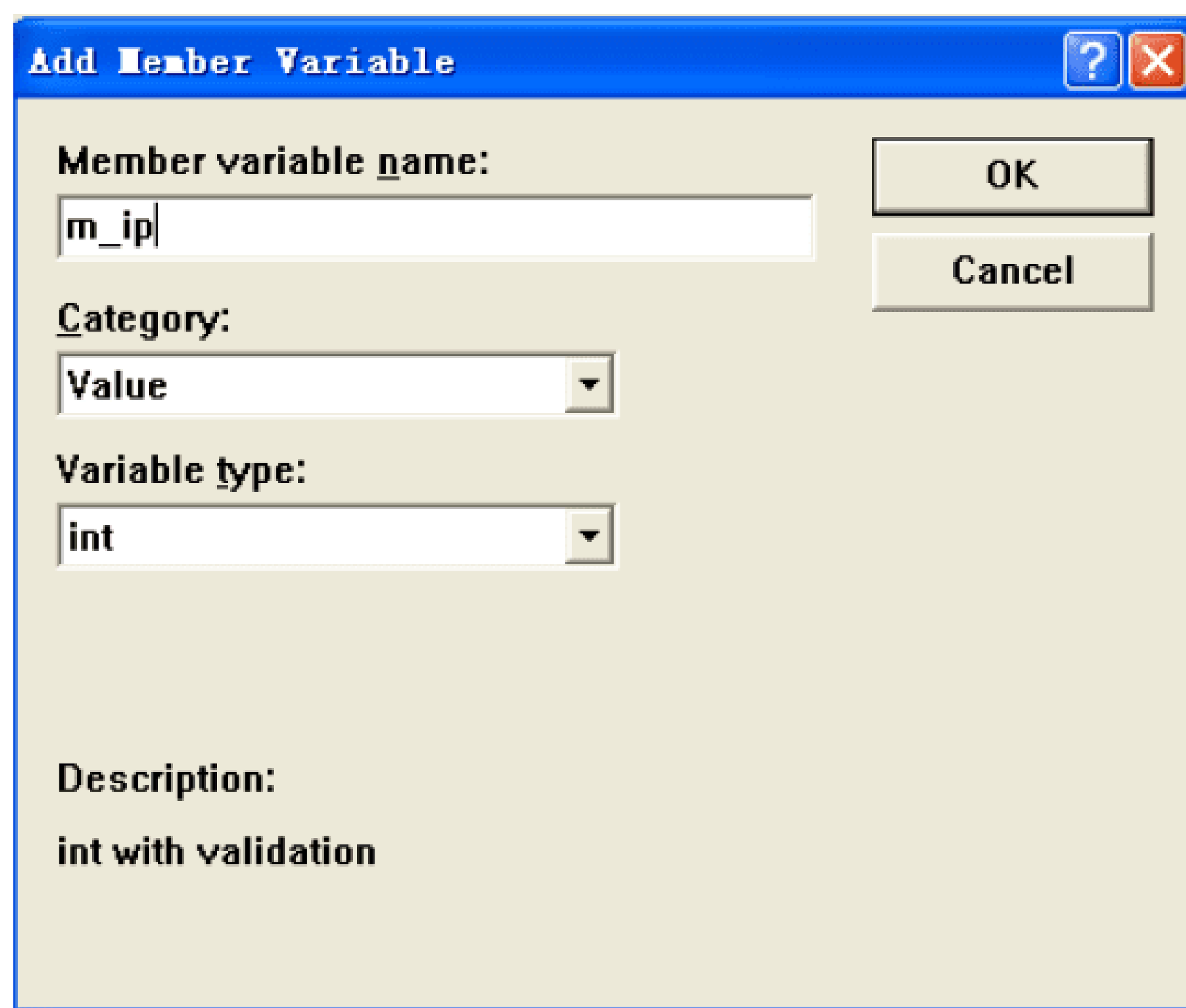


图 11.12　Add Member Variable 对话框

在该对话框中，用户在第一个编辑框中输入控件变量的名称，当选择好变量类型后，单击 OK 按钮添加控件变量成功。

注意：由于在消息对话框程序中，添加控件变量的方法都是一样的。所以，为其他控件关联变量的方法也是一样，在本节中不再向用户进行赘述。

通过本节，用户基本完成了服务器界面的初始化工作。关于其他功能的实现，将在后面的内容中向用户进行讲述。

11.1.2　客户端

在客户端实例中，实例程序的界面应该与 QQ 的界面相似。因此，在 VC 中新建一个工程，修改工程名为“仿 QQ 程序客户端”。而其他设置均与服务器端一样。

1. 客户端界面设计

用户可以在对话框资源编辑器中，将窗口拖放到与 QQ 窗口同样大小，如图 11.13 所示。但是，用户在实际使用该实例时，常常需要使用鼠标将窗口拉大或者是拉小。为了使程序运行时具有该功能，用户可以右击对话框面板，在弹出的快捷菜单中选择“属性”命令，弹出“对话 属性”对话框，如图 11.14 所示。

用户在该对话框中，选择“样式”选项卡，在边框下拉列表框中选择“调整大小”选项。当客户端实例运行时，用户便可以使用鼠标对窗口进行缩放了。请用户参考随书光盘中的实例代码。

现在，用户可以在 VC 的资源管理器中，为客户端主窗口添加相应的功能控件。添加控件后的界面效果如图 11.15 所示。

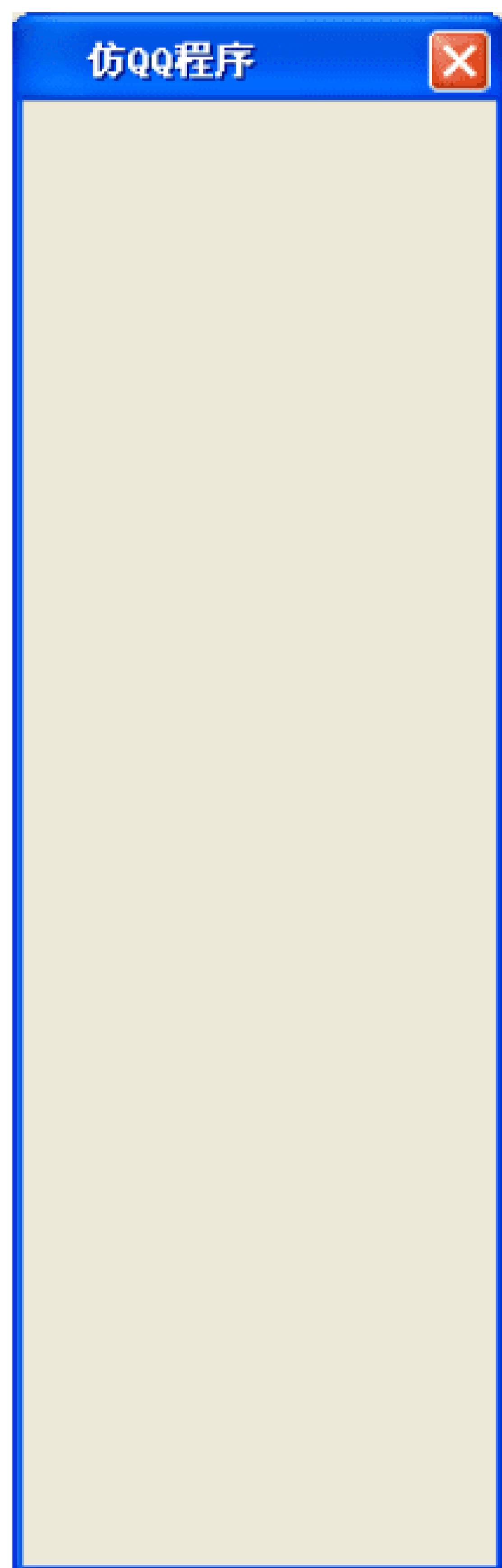


图 11.13　客户端窗口初始化大小

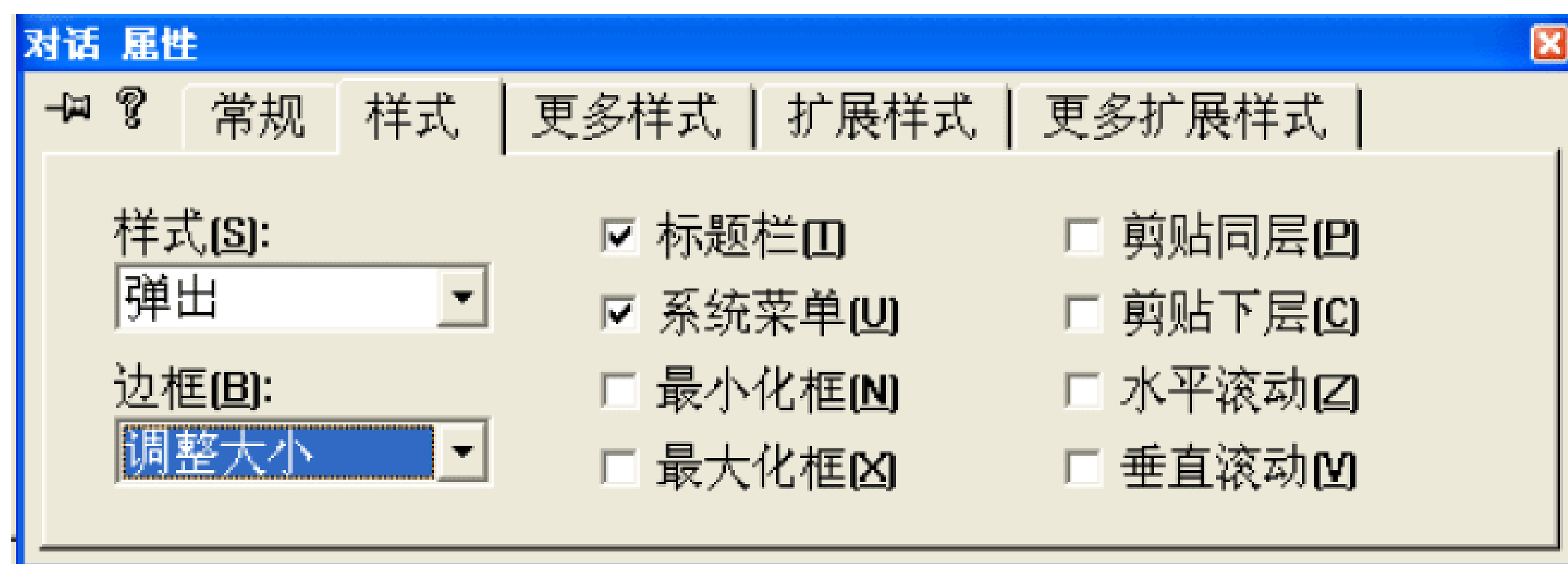


图 11.14　“对话 属性”对话框

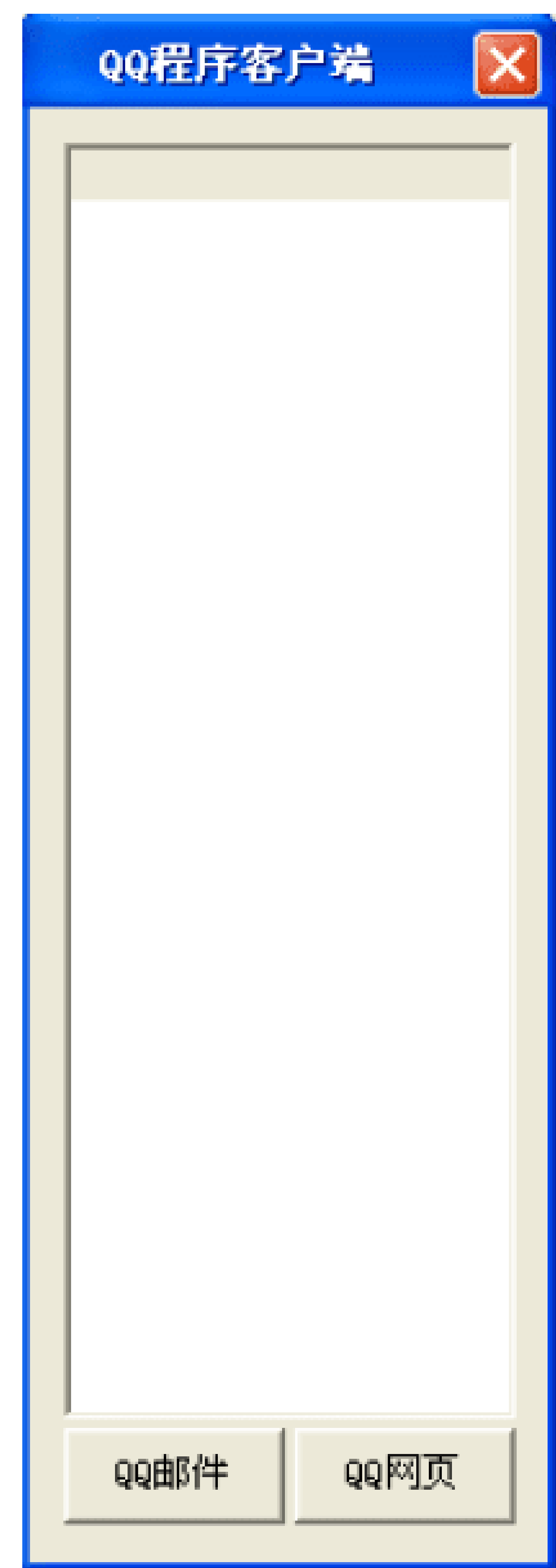


图 11.15 添加控件后的客户端界面

其中，添加的控件 ID、类型以及作用等如表 11.2 所示。

表 11.2　控件ID、类型以及作用

控　件　ID	控 件 类 型	控 件 作 用
IDC_LIST1	列表控件	显示好友
IDC_EMAIL	按钮	QQ邮件
IDC_NET	按钮	QQ网页

用户添加控件成功以后，为了使添加的控件能随窗口的大小改变而改变。该功能应该在窗口类的函数 OnSize()中实现，代码如下：

```
void CQQDlg::OnSize(UINT nType, int cx, int cy)
{
    CDialog::OnSize(nType, cx, cy);
    RECT rect;                                          //定义结构体变量
    ::GetWindowRect(m_hWnd,&rect);                      //获取窗口的大小
    GetDlgItem(IDC_LIST1)->MoveWindow(&rect,true); //根据窗口的大小移动控件
   GetDlgItem(IDC_EMAIL)->MoveWindow(&rect,true);
   GetDlgItem(IDC_NET)->MoveWindow(&rect,true);
}
```

当窗口的大小发生变化时，程序会调用该函数进行处理。通过以上代码，用户实现了控件随窗口的大小改变而改变。

2. 显示QQ图标

为了使实例程序与真实 QQ 的效果一样，用户可以在程序中添加一个 QQ 图标。其方法如下：

（1）在实例工程中，添加一个图标资源。选择“插入”|“资源”命令，弹出“插入资源”对话框，如图 11.16 所示。



图 11.16　插入资源对话框

用户选中资源类型列表中的选项 Icon，再单击“引入”按钮，选择并插入图标文件即可。在 VC 资源管理器的图标项中，用户可以查看刚添加的图标文件，如图 11.17 所示。

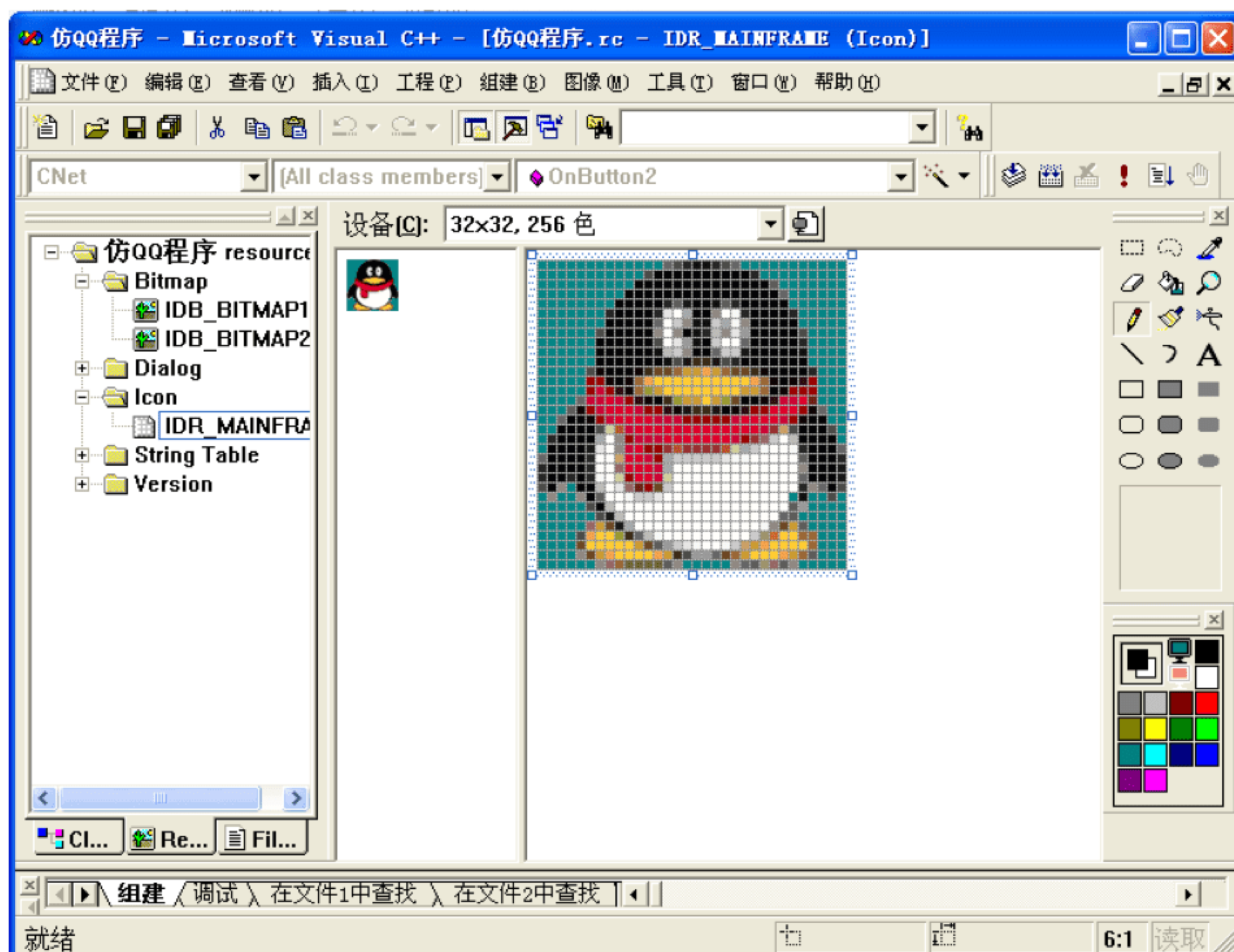


图 11.17　刚插入的图标资源

（2）用户可以将该图标文件的名称修改为 IDR_MAINFRAME，并且在对话框类 CQQDlg 的初始化函数中调用函数载入该图标资源到程序中。代码如下：

```
CQQDlg::CQQDlg(CWnd* pParent /*=NULL*/)
```

```
    : CDialog(CQQDlg::IDD, pParent)                    //对话框类的初始化函数
{
    m_hIcon = AfxGetApp()->LoadIcon(IDR_MAINFRAME);//载入图标资源
}
```

（3）在对话框的初始化函数 OnInitDialog()中，调用函数 SendMessage()向主窗口发送消息 WM_SETICON 设置该对话框的图标。代码如下：

```
BOOL CQQDlg::OnInitDialog()                          //对话框初始化函数
{
    CDialog::OnInitDialog();
    SetIcon(m_hIcon, TRUE);
    SetIcon(m_hIcon, FALSE);
    ::SendMessage(this->m_hWnd,WM_SETICON,0,(unsigned int)m_hIcon);
                                                     //向对话框发送设置图标的消息
    return TRUE;
 }
```

（4）用户保存，并且编译运行程序，结果如图 11.18 所示。

到这里为止，用户已经成功地为应用程序添加了 QQ 图标。

3．列表控件初始化

当实例程序启动时，列表控件的初始化工作主要是显示列表控件的标题以及好友列表。所以，用户需要在实例工程中插入位图资源，并将其命名为 IDB_BITMAP1，如图 11.19 所示。

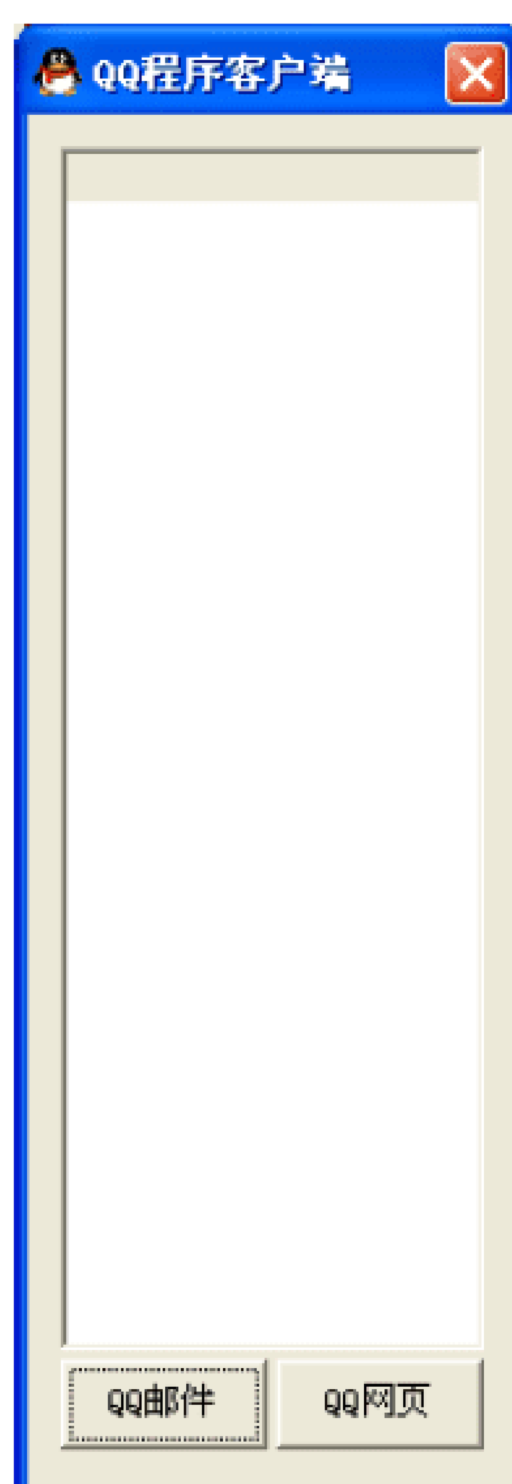


图 11.18　设置实例程序的图标

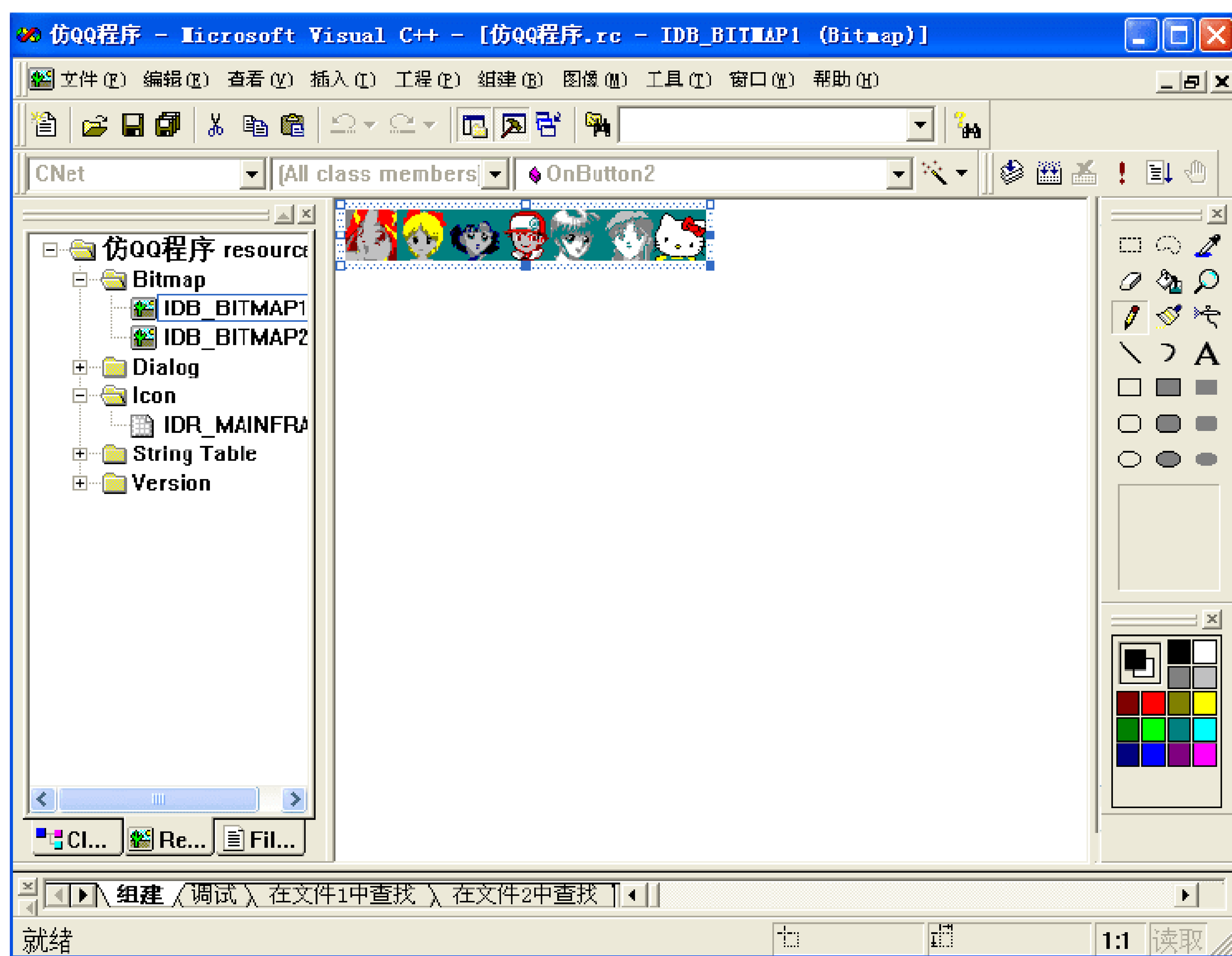


图 11.19　插入的位图资源

位图资源插入成功后，用户便可以在对话框初始化函数中添加代码。代码如下：

```
BOOL CQQDlg::OnInitDialog()
{
    CDialog::OnInitDialog();
    ...                                              //省略部分代码
    CImageList *img;                                 //定义图像列表指针
    img=new CImageList();                            //创建图像列表对象
```

```
    img->Create(32,32,ILC_COLOR4,3,2);
    img->Add(&m_bit,RGB(255,0,255));         //向图像列表中添加图像
    LVCOLUMN lv;                             //定义列表结构体变量
lv.mask=LVCF_TEXT|LVCF_FMT|LVCF_WIDTH|LVIF_IMAGE;
                                             //初始化该结构体中的成员
lv.fmt=LVCFMT_CENTER;
lv.pszText="      好友列表";                 //设置列表标题
lv.cx=140;                                   //指定该列宽度
m_list.InsertColumn(0,&lv);                  //插入列表标题
m_list.SetImageList(img,LVSIL_SMALL);        //设置图像列表
CString str[7];
for(int i=0;i<7;i++)
{
recv(s,str[i].GetBuffer(1),sizeof(str[i]),NULL);//接收服务器发送的好友列表
m_list.InsertItem(i,str[i],i);               //向列表控件中添加项目
}
return TRUE;
}
```

在代码中，函数 SetImageList()的作用是将图像列表以指定方式显示到列表控件中。其原型如下：

```
CImageList* SetImageList( CImageList* pImageList, int nImageList );
```

该函数有两个参数。其中，参数 pImageList 指向 CImageList 类的指针；参数 nImageList 表示显示的方式，其值如表 11.3 所示。

表 11.3　图像列表在列表控件中的显示方式

取　　值	意　　义
LVSIL_NORMAL	以大图显示
LVSIL_SMALL	以小图显示
LVSIL_STATE	以固定大小进行显示

用户将上面的代码编译、运行后，结果如图 11.20 所示。



图 11.20　在列表中显示位图资源

4．设置对话框背景

用户可以从资源文件中，插入一幅位图作为程序界面的背景，将资源 ID 修改为 IDB_BITMAP2。首先，在对话框类中声明设备句柄 dc1。代码如下：

```
class CQQDlg : public CDialog
{
public:
    ...                                                  //省略部分代码
    HDC dc1;                                             //声明设备句柄
 }
```

然后，在函数 OnPaint()中，创建一个与对话框设备环境相兼容的 DC 句柄并返回。利用这个返回的兼容设备 DC，用户便可以将载入的位图资源放入该兼容 DC 中。最后，调用函数 StretchBlt()将兼容 DC 中的位图资源复制到对话框的实际 DC 中显示。代码如下：

```
void CQQDlg::OnPaint()
{
...                                                  //省略部分代码
m_bit1.LoadBitmap(IDB_BITMAP2);                      //载入位图资源
dc1=::CreateCompatibleDC(::GetDC(this->m_hWnd));     //创建兼容 DC
::SelectObject(dc1,m_bit1.m_hObject);                //选取位图到兼容 DC 中
::StretchBlt(::GetDC(this->m_hWnd),0,0,500,650,dc1,0,0,300,300,SRCCOPY)
;
                                                     //复制位图资源
}
```

在上面的代码中，函数 StretchBlt()的作用是将兼容 DC 中的位图资源直接复制到对话框的 DC 中。该函数的最后一个参数 SRCCOPY 表示以复制方式进行传送。用户将代码进行保存、编译并运行，如图 11.21 所示。



图 11.21 设置背景图片的对话框

在这里，用户可以参考随书光盘中实例代码，并且可以试着修改代码实现控件的透明化。这样将有利提高用户的学习质量。

5. 添加消息发送对话框

用户在使用 QQ 进行网络通信时，当双击列表中的项目时，都会弹出一个消息发送对话框。因此，在本实例中，用户实现这一功能需要在工程中添加一个对话框，并关联对应的新类。

首先，选择“插入”|“资源”命令，弹出“插入资源”对话框，如图 11.5 所示。用户在该对话框中，选择资源类型为 Dialog 后，单击“新建”按钮。接着，用户在 VC 资源管理器中，编辑新建的消息发送对话框以实现实际功能，如图 11.22 所示。

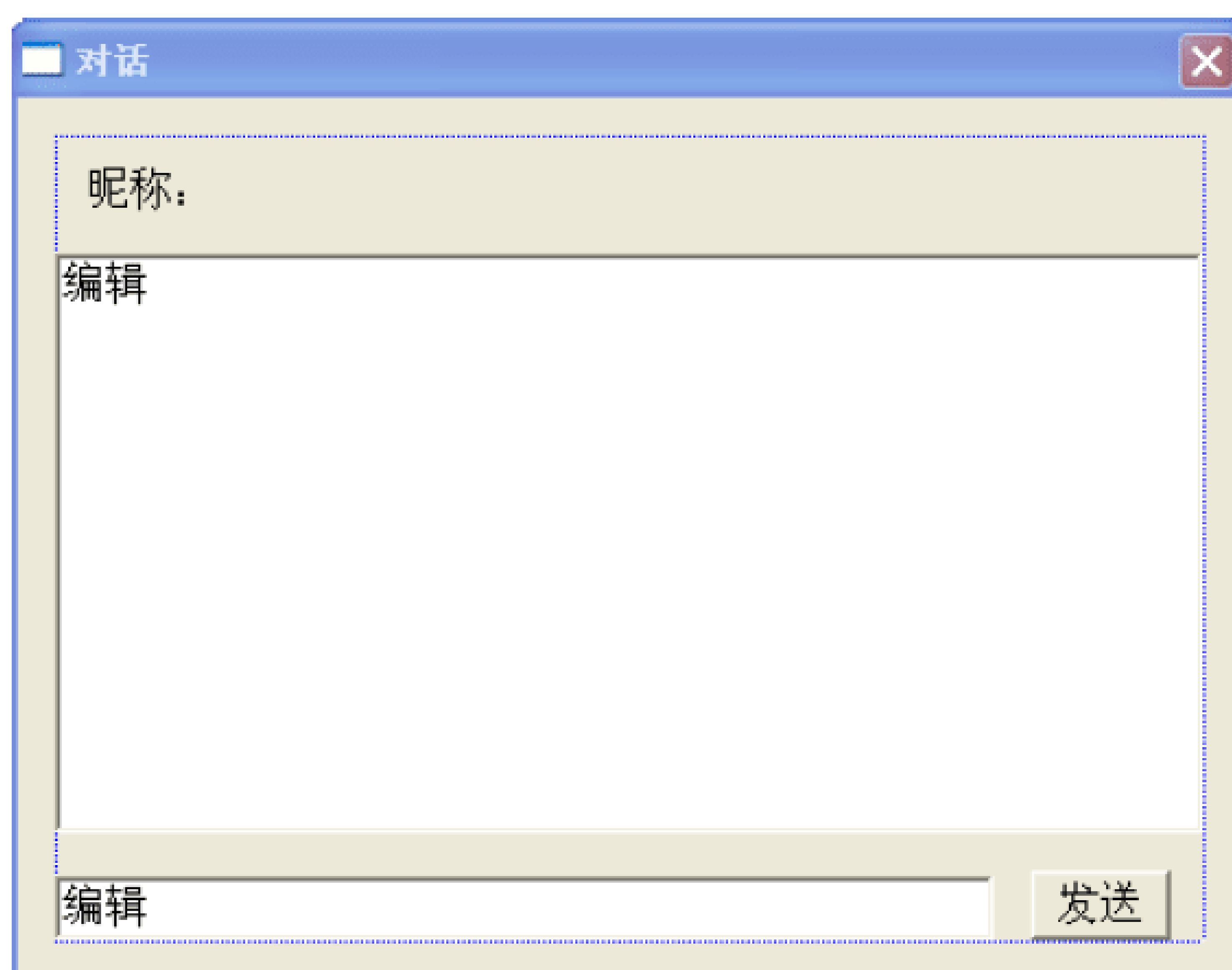


图 11.22　消息发送对话框界面

然后，用户还必须为该对话框关联一个新类才能在实例程序中使用该对话框。用户可以使用快捷键 Ctrl+W 弹出 MFC 应用程序向导。此时，编译器会弹出 Adding a Class 对话框，如图 11.23 所示。

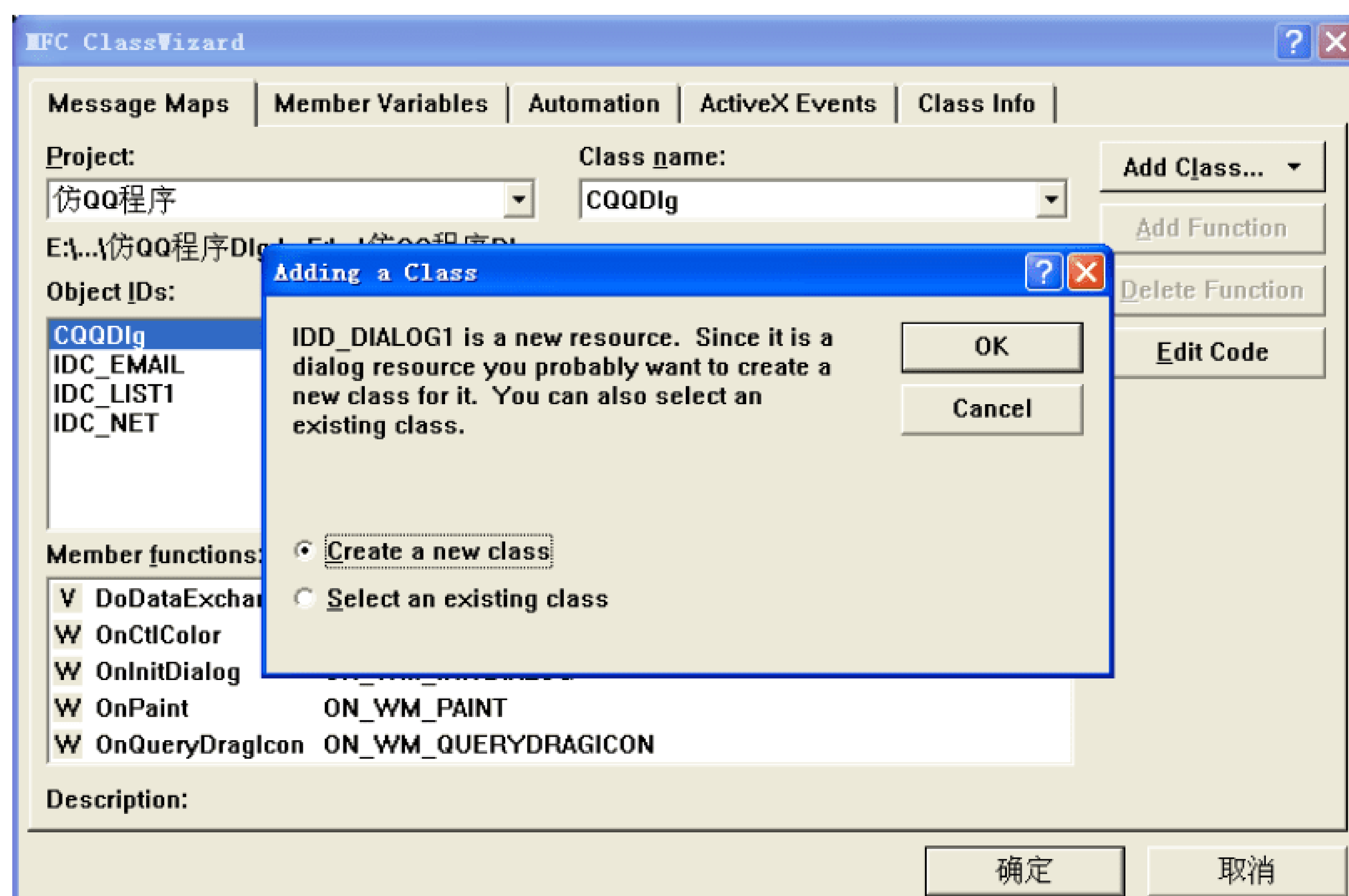


图 11.23　“Adding a Class”对话框

在图 11.23 中，用户选择 Create a new class 单选按钮后，单击 OK 按钮，弹出 New Class 对话框，如图 11.24 所示。

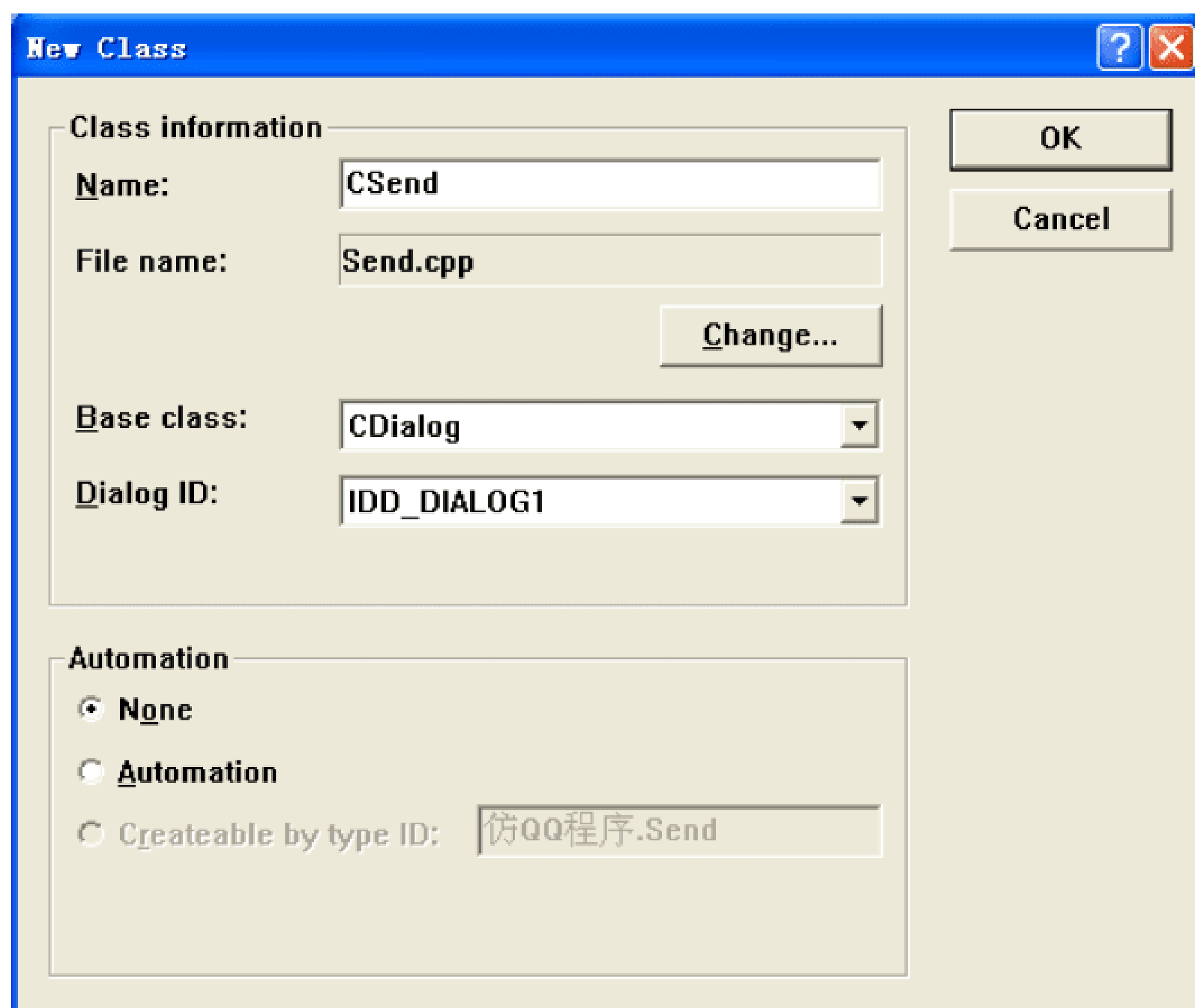


图 11.24　New Class 对话框

在该对话框中，用户可以修改新类名及其基类名等类信息。单击 OK 按钮完成新类的创建。

注意：用户在实例程序中，使用该对话框前，必须在程序中包含该对话框类的头文件，并且需要在实例程序中定义该对话框类的实例对象。

本节主要向用户讲解了实例程序中，服务器与客户端界面的设计以及消息发送对话框界面的设计等。

11.2　通信数据

在本节中，将向用户讲解关于服务器与客户端之间传输的通信数据格式和主要功能的实现方法。最后，将这些功能以及数据封装在 CData 类中供用户使用。

11.2.1　定义通信数据结构

一般情况下，在服务器与客户端之间的通信数据必须具有相同的数据结构。这样，既能保证数据的完整性，又能使这些数据具有良好的可读性。

首先，用户需要分析该实例中，对于通信数据而言，哪些是比较重要的。在本章中向用户列举了一些数据，例如，客户端 IP 地址、客户昵称等。

然后，用户需要将这些数据组合到一个结构体中。代码如下：

```
typedef struct qq_str{            //自定义 QQ 数据结构体
```

```
char  m_name[50];                //发送信息客户昵称
char  m_name1[50];               //接收信息客户昵称
int  m addr;                     //客户端 IP 地址
char  msg[1024];                 //客户端与服务器的通信内容
} qqstruct;
```

上面的结构体 qqstruct 已经基本包含了本实例中客户端的主要数据。当然，用户在实际使用该结构时，在服务器端同样需要该结构体。所以，用户在构造服务器端数据结构时，需要首先定义一个该结构变量。

另外，服务器端为了将接收到的消息转发到该信息真正的接收方，必须定义第二个结构体。在该结构体中包含有结构体 qqstruct 变量和服务器端调用函数 accept()时所返回的通信套接字。结构如下：

```
typedef struct q_str{            //服务器转发消息数据结构
   qqstruct qq;                  //客户端数据结构变量
   SOCKET s;                     //消息接收方套接字句柄
}qqmsg;
```

注意：在服务器端转发消息时，是利用接收信息方客户的昵称查找相应的通信套接字的。

11.2.2　功能实现

在本节中，将向用户讲解服务器端以及客户端的各个功能函数的实现方法。在这些功能函数中，对数据结构的使用同样也是非常重要的一个知识点。

1．服务器功能实现

首先，当服务器启动时会在指定端口等待客户端的连接。如果连接成功，则将在服务器端列表控件中显示客户端的相关信息。代码如下：

```
void CData::bind()
{
    WSADATA data;                                    //定义结构体变量
   CString name;                                     //定义主机名字符串
    DWORD ss=MAKEWORD(2,0);                          //指定套接字版本
   ::WSAStartup(ss,&data);                           //初始化套接字库
   s=::socket(AF_INET,SOCK_STREAM,IPPROTO_TCP);
                                                     //创建套接字
   sockaddr_in addr;                                 //定义地址结构变量
::gethostname((char*)&name,(int)sizeof(name));       //获得主机名字
    hostent *p=::gethostbyname((char*)&name);        //从主机名获取主机地址
     in_addr *a=(in_addr*)*p->h_addr_list;           //获得本机 IP 地址
     CString str14=::inet_ntoa(a[0]);                //转换字符串 IP 地址
   addr.sin_family=AF_INET;                          //填充地址结构
    addr.sin_port=htons(80);                         //指定监听端口为 80
    addr.sin_addr.S_un.S_addr=inet_addr(str14);//指定主机 IP 地址
    ::bind(s,(sockaddr*)&addr,sizeof(addr));         //将本地信息绑定到套接字
    ::listen(s,5);                                   //监听
WSAAsyncSelect(s,this->m_hWnd,WM_SOCK,FD_ACCEPT|FD_READ);
```

```
                                                    //将套接字设置为异步模式
}
```

在程序中，用户使用函数 gethostname()获取本地主机名，并且通过该主机名得到相应的 IP 地址信息。再将本地主机的相关信息绑定到创建的套接字上，并将该套接字设置为异步模式。

注意：在本节中向用户讲解的各个功能函数均为自定义类 CData 的成员函数。关于该类的具体封装方法将在 11.2.3 节中具体讲解。

然后，用户在程序中必须定义与套接字消息相应的响应函数，并且使用消息映射宏将套接字消息 WM_SOCK 与其响应函数相关联才能实现套接字的异步模式。用户为套接字消息添加消息映射，代码如下：

```
BEGIN_MESSAGE_MAP(CQQDlg, CDialog)
    //{{AFX_MSG_MAP(CQQDlg)
    ON_WM_SYSCOMMAND()
    ON_WM_QUERYDRAGICON()
    ...                                     //省略部分消息映射项
    ON_MESSAGE(WM_SOCK,Onsoc)               //套接字消息映射
    ON_WM_CTLCOLOR()
    ON_WM_TIMER()
    ON_WM_LBUTTONDOWN()
    //}}AFX_MSG_MAP
END_MESSAGE_MAP()
```

在消息映射宏中，用户可以看到与套接字消息相关联的函数是 Onsoc()。该函数的声明代码如下：

```
class CQQDlg : public CDialog
{
public:
    ...                                                  //省略部分代码
afx_msg void Onsoc(WPARAM wParam,LPARAM lParam);         //声明套接字消息响应函数
}
```

用户对套接字消息响应函数声明以后，便可以在程序中实现该函数。代码如下：

```
int i=0;                                            //定义全局变量
qqstruct qq;                                        //定义结构体 qqstruct 变量
qqmsg msg[5];                                       //定义结构体 qqmsg 变量
void CQQDlg::Onsoc(WPARAM wParam,LPARAM lParam)
{
    switch (lParam)                                 //判断套接字事件
{
      case FD_ACCEPT:                               //套接字连接事件
            {
        msg[i].s=::accept(s,NULL,NULL);             //保存客户端连接时返回的套接字
          if(i<5)                                   //判断连接的客户端是否已满
{
i+=1;
   }
else
{
i=0;
}
```

```
            }
            break;
        case FD_READ:                               //套接字读取事件
              {
            recv(msg[i].s, msg[i].qq,sizeof(msg[i].qq),NULL);
                                                    //接收相应套接字上的消息
int nRow=m_list.InsertItem(m_list.GetItemCount()+1, msg[i].qq.m_name);
                                                    //向列表控件中插入行数据
m_list.SetItemText(nRow,1, msg[i].qq .m_addr); //设置数据
              }
            break;
        default:  break;
}
}
```

保存并编译运行以上代码，当客户端向服务器发送连接请求后，服务器同意连接并将客户端的相关信息显示在列表控件中，如图 11.25 所示。

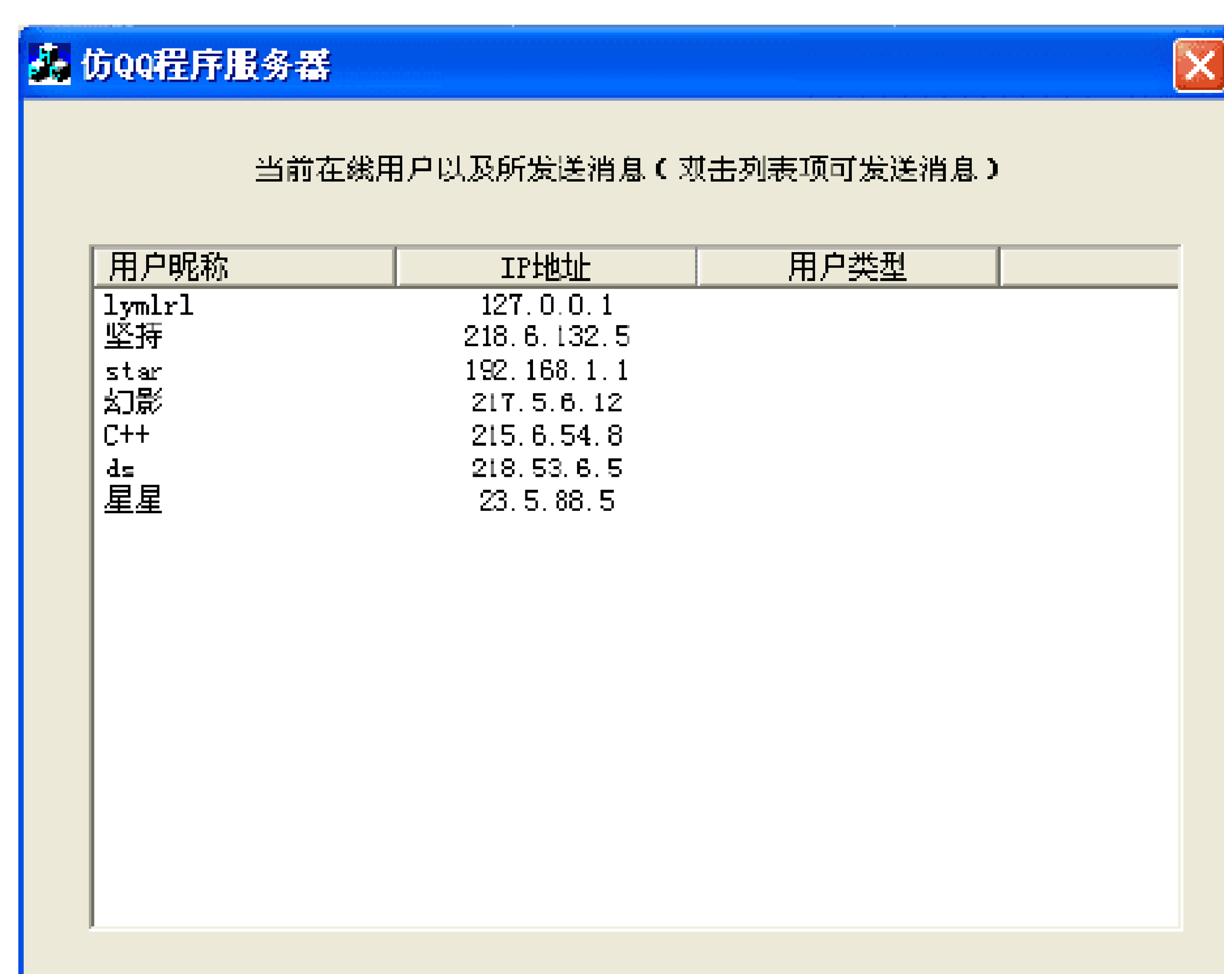


图 11.25　服务器显示连接客户端的相应信息

当服务器端接收客户端所发送的消息以后，服务器还应该将该消息转发到客户端指定的接收方。因此，在服务器套接字的消息响应函数中，还需要添加相应的代码，其代码如下：

```
{
...                                                     //省略部分代码
for(int i=0;i<5;i++)
{
  if(msg[i].qq.m_name1== m_list.GetItemText(i,3))   //判断昵称是否存在
{
send(msg[i].s,&msg[i].qq.msg,1024,NULL);            //将信息转发到接收方
  }
}
...                                                     //省略部分代码
}
```

同时，用户也可以双击列表中指定客户端的昵称，向该客户端发送信息。该功能在列表控件的双击消息响应函数 OnDblclkList1()中实现。代码如下：

```
void CQQDlg::OnDblclkList1(NMHDR* pNMHDR, LRESULT* pResult)
{
POSITION pos=m list.GetFirstSelectedItemPosition();//获取鼠标双击位置
if(pos==NULL)                                           //判断位置是否为空
{
```

```
    MessageBox("用户双击的位置错误或该列表为空!");
}
else
{
        int nItem=m_list.GetNextSelectedItem(pos);
                                                    //获取该位置的索引值
   CString  str=m_list.GetItemText (nItem,1);  //获取相应的客户号码或昵称
CString  str1=m_list.GetItemText (nItem,2);
CMessage mesg;                                      //定义消息发送框对象
mesg.m_num=str;                                     //对话框初始化时赋值
mesg.m_ip=str1;
mesg.DoModal( );                                    //显示消息发送框
}
    *pResult = 0;
}
```

当用户双击列表控件中的某一项时，会弹出消息发送对话框，如图 11.26 所示。

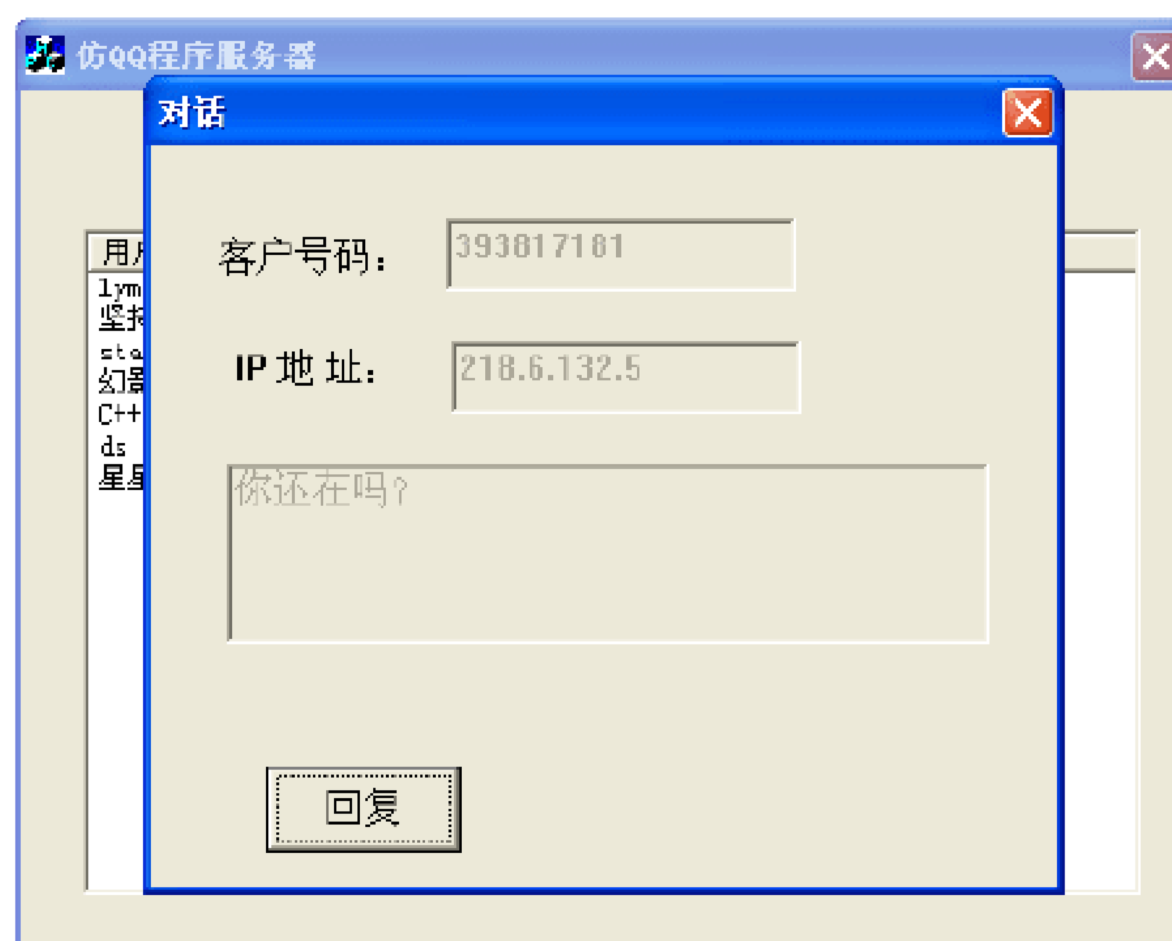


图 11.26　显示消息发送对话框

在消息发送对话框中，用户单击“回复”按钮，程序会显示信息输入框以及“发送”功能按钮。代码如下：

```
void CMessage::OnRelay()                            //“回复”按钮消息响应函数
{
    GetDlgItem(IDC_TEXT2)->ShowWindow(true);    //使按钮控件可见
    GetDlgItem(IDC_SEND)->ShowWindow(true);
}
```

运行上面的程序，用户单击“回复”按钮后，消息发送对话框会显示信息发送编辑框以及“发送”按钮，如图 11.27 所示。

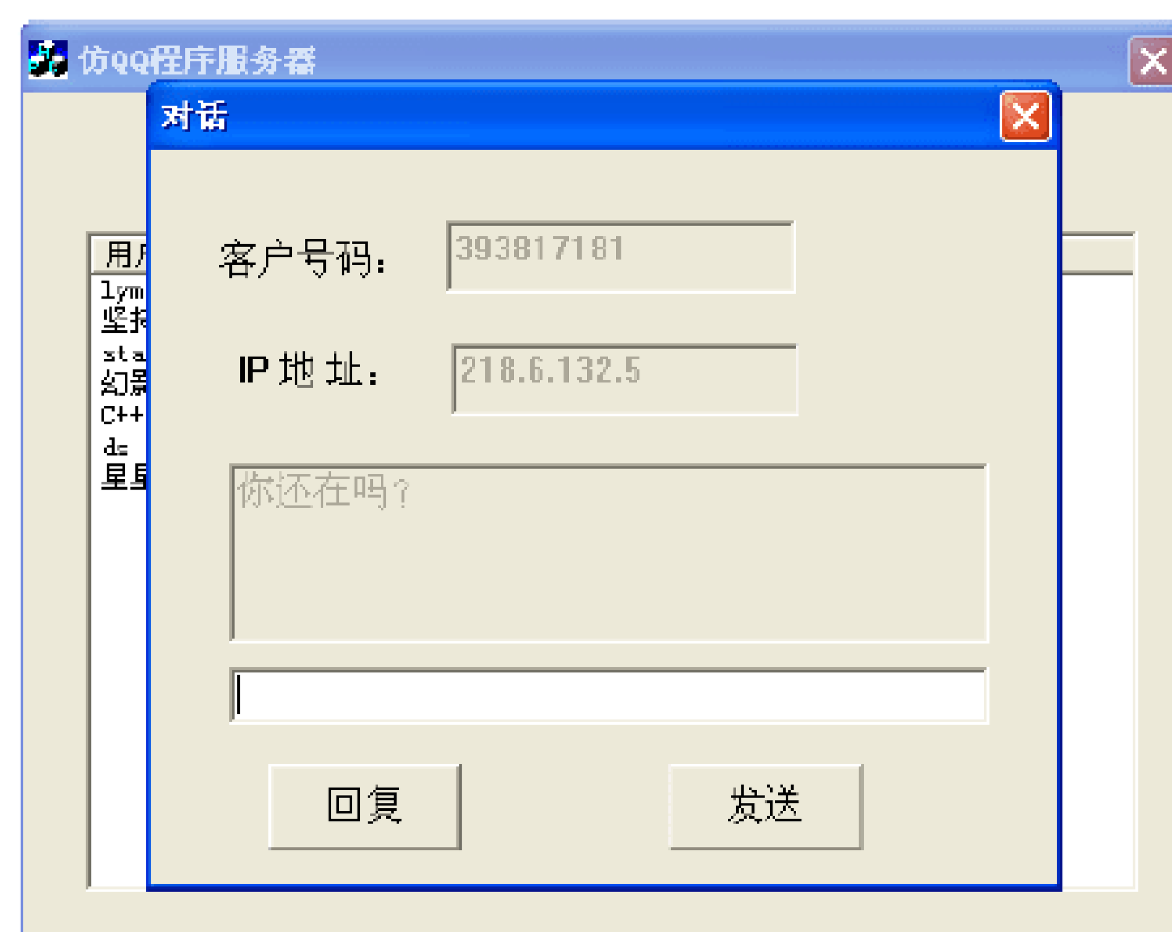


图 11.27　显示编辑框和“发送”按钮

其中，“发送”按钮的消息响应函数功能是将用户编辑的信息发送到指定客户端。代码如下：

```
void CMessage::OnSend()
{
CString msg,str;                                    //定义字符串变量
GetDlgItem(IDC_TEXT2)->GetWindowText(msg);          //获取消息编辑框中的内容
GetDlgItem(IDC_IP)->GetWindowText(str);             //获取显示的 IP 地址
for(int i=1;i<=m_list.GetItemCount();i++)           //循环获取列表控件中的 IP 地址
{
    CString m_ip;                                   //定义字符串变量
    m_list.GetItemText(i,m_ip);                     //获取列表项目中的 IP 地址
    if(m_ip==str)                                   //若 IP 地址相同
    {
    send(msg[i].s,msg.GetBuffer(1),sizeof(msg),NULL);//发送信息到客户端
    }
}
}
```

在服务器端，用户已经实现了一些常用的功能。当用户学习本节知识时，需要将书中的理论知识与实例代码相结合，这样学习效果将比较明显。

2. 客户端功能实现

在实例工程中，用户仅需要完成客户端向服务器端发送相关信息即可。首先，使用 MFC 应用程序向导为列表控件添加双击的消息响应函数 OnDblclkList1()。然后，在该函数中显示消息发送对话框发送消息。代码如下：

```
void CQQDlg::OnDblclkList1(NMHDR* pNMHDR, LRESULT* pResult)
{
POSITION pos=m_list.GetFirstSelectedItemPosition();//获取双击位置
if(pos==NULL)                                             //判断位置是否为空
{
    MessageBox("用户双击的位置错误或该列表为空!");
}
else
{
        int nItem=m_list.GetNextSelectedItem(pos);
                                                    //获取该位置的索引值
  CString  str=m_list.GetItemText(nItem,0);         //获取相应的客户号码或昵称
send.m_name="客户昵称：";                           //对话框初始化时赋值
strcat(send.m_name.GetBuffer(1),str.GetBuffer(1));//连接字符串
mesg.DoModal( );                                    //显示消息发送框
}
    *pResult = 0;
}
```

用户编译并运行以上代码后，鼠标双击客户端列表的某一项，程序将弹出消息发送对

话框，如图 11.28 所示。

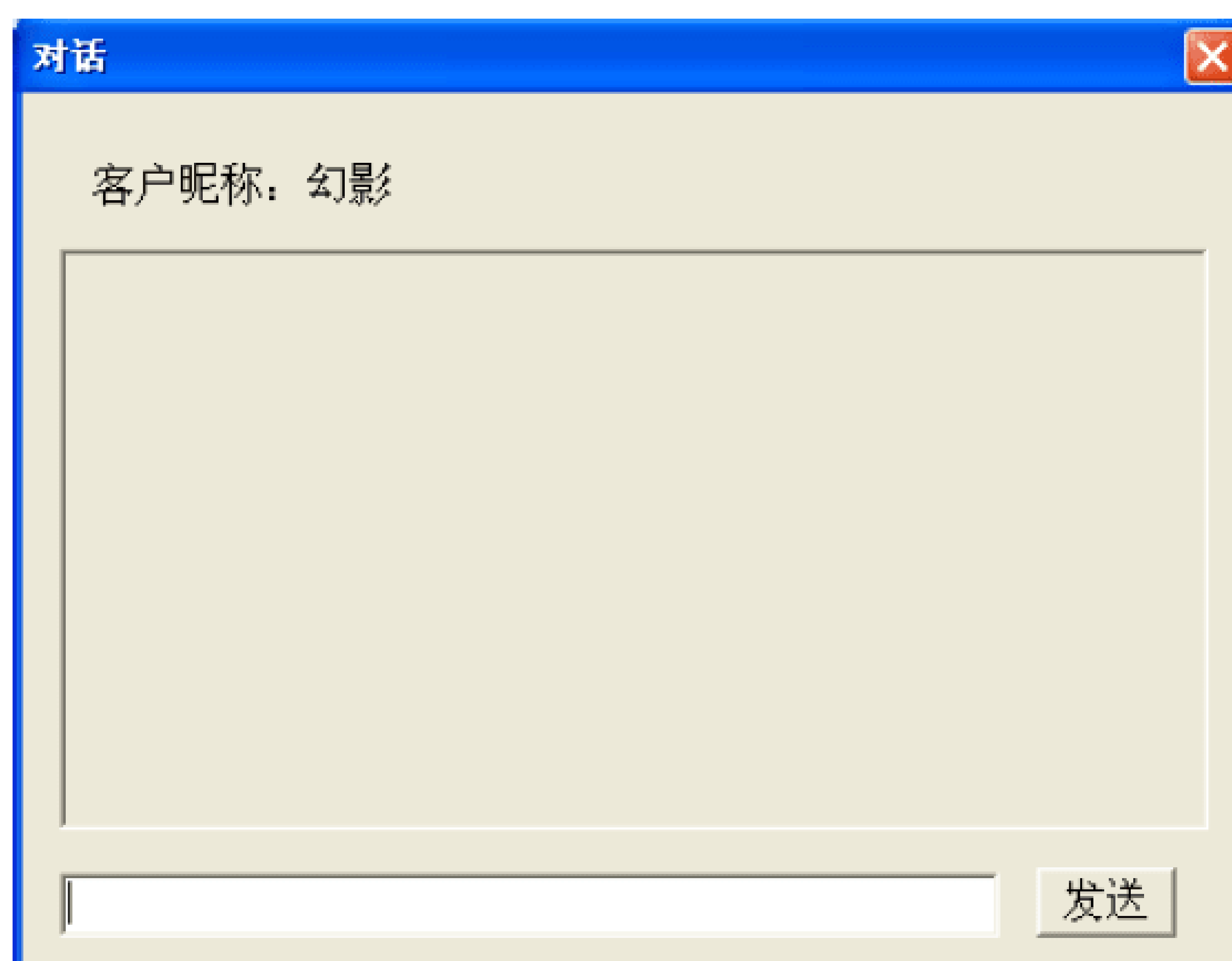


图 11.28　弹出消息发送对话框

当程序弹出消息发送对话框后，用户便可以输入相应信息，单击“发送”按钮，将该信息发送到服务器端进行转发。代码如下：

```
void CSend::OnSend()                                    //发送按钮消息响应函数
{
    CString str,str1;                                   //定义字符串变量
sockaddr_in addr;                                       //定义地址结构变量
 addr.sin_family=AF_INET;                               //填充地址结构
addr.sin_port=htons(80);                                //指定监听端口为 80
addr.sin addr.S un.S addr=inet addr(str14);             //指定主机 IP 地址
 connect(s,( sockaddr*)addr,sizeof(addr),NULL);//连接服务器
GetDlgItem(IDC_EDIT1)->GetWindowText(str);              //获取用户输入的数据
send(s,str.GetBuffer(1),sizeof(str),NULL);              //发送信息
GetDlgItem(IDC_EDIT2)->GetWindowText(str1);             //获取信息显示框中的内容
str1+="\r\n";                                           //添加回车换行符
str1+=str;                                              //连接字符串
GetDlgItem(IDC_EDIT2)->SetWindowText(str1);             //设置信息显示框的内容
}
```

用户将上面的代码保存、编译并运行后，便可以实现客户端向服务器端发送消息，如图 11.29 所示。

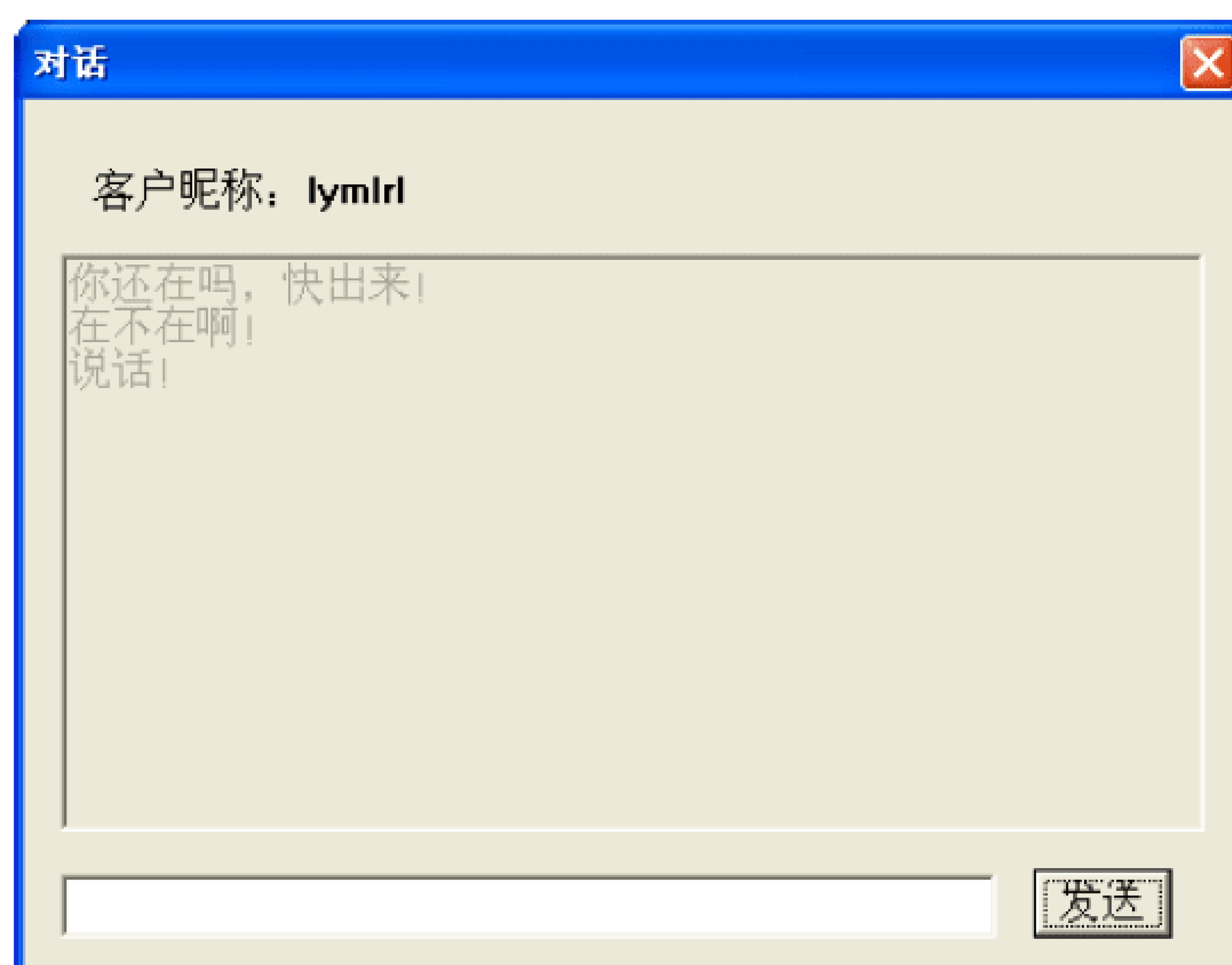


图 11.29　消息发送对话框发送消息

注意： 用户在学习过程中，可以在随书光盘的实例代码中，试着添加自己的代码。例如，为消息发送对话框设置背景等。这样，能促进用户的学习。

11.2.3　封装 CData 类

在 11.2.2 节中，分别向用户介绍了服务器端和客户端的各个功能实现方法。为了让用户深入了解 C++面向对象编程的方法，在本节中，将各个功能函数封装在 CData 类中。使用时，直接生成 CData 类实例对象，通过该对象调用相关的功能函数。

1．定义CData类

首先，用户需要定义 CData 类头文件 Data.h。代码如下：

```
class CData
{
public:
    CData();                                      //构造函数
   void bind();                                   //服务器绑定套接字函数
   void sevsend(SOCKET&,char*,int,int);           //服务器发送函数
   void sevrecv(SOCKET&,char*,int,int);           //服务器接收函数
   void connect();                                //客户端连接函数
   void send(SOCKET&,char*,int,int);              //客户端发送函数
   void recv(SOCKET&,char*,int,int);              //客户端接收函数
~CData();
   };
```

然后，在定义文件 Data.cpp 中，实现各个功能函数的具体方法。代码如下：

```
#include "Data.h"
#include "iostream.h"
CData::CData()                                        //构造函数
{
   }
void CData::bind()                                    //服务器绑定套接字函数
{
  WSADATA data;                                       //定义结构体变量
   CString name;                                      //定义主机名字符串
    DWORD ss=MAKEWORD(2,0);                           //指定套接字版本
   ::WSAStartup(ss,&data);                            //初始化套接字库
   s=::socket(AF_INET,SOCK_STREAM,IPPROTO_TCP);
                                                      //创建套接字
   sockaddr_in addr;                                  //定义地址结构变量
::gethostname((char*)&name,(int)sizeof(name));        //获得主机名字
    hostent *p=::gethostbyname((char*)&name);         //从主机名获取主机地址
     in_addr *a=(in_addr*)*p->h_addr_list;            //获得本机 IP 地址
     CString str14=::inet_ntoa(a[0]);                 //转换字符串 IP 地址
   addr.sin_family=AF_INET;                           //填充地址结构
    addr.sin_port=htons(80);                          //指定监听端口为 80
    addr.sin_addr.S_un.S_addr=inet_addr(str14);//指定主机 IP 地址
    ::bind(s,(sockaddr*)&addr,sizeof(addr));          //将本地信息绑定到套接字
    ::listen(s,5);                                    //监听
WSAAsyncSelect(s,this->m_hWnd,WM_SOCK,FD_ACCEPT|FD_READ);
```

```
                                                     //将套接字设置为异步模式
}
void CData::sevsend(SOCKET s,char* buff,int len,int flags)
                                                     //自定服务器发送函数
{
            send(s,buff,len,flags);                  //调用 API 发送函数
}
void CData::sevrecv(SOCKET s,char* buff,int len,int flags)
                                                     //自定义服务器接收函数
{
 recv(s,buff,len,flags);                             //调用 API 接收函数
  }
void CData::connect()
{
sockaddr_in addr;                                    //定义地址结构变量
 addr.sin family=AF INET;                            //填充地址结构
addr.sin_port=htons(80);                             //指定监听端口为 80
addr.sin addr.S un.S addr=inet addr(str14);          //指定主机 IP 地址
 if(connect(s,( sockaddr*)addr,sizeof(addr),NULL)==-1)
                                                     //连接服务器
{
  MessageBox("连接服务器失败！");                    //提示用户连接服务器失败
  }
}
void CData::send(SOCKET s,char* buff,int len,int flags)
                                                     //自定客户端发送函数
{
  GetDlgItem(IDC EDIT1)->GetWindowText(str);
                                                     //获取用户输入的数据
send(s,str.GetBuffer(1),sizeof(str),NULL);
                                                     //发送信息
GetDlgItem(IDC_EDIT2)->GetWindowText(str1);
                                                     //获取信息显示框中的内容
str1+="\r\n";                                        //添加回车换行符
str1+=str;                                           //连接字符串
GetDlgItem(IDC_EDIT2)->SetWindowText(str1);
                                                     //设置信息显示框的内容
}
void CData::recv(SOCKET s,char* buff,int len,int flags)
                                                     //自定义客户端接收函数
{
  char buff[1024];                                   //定义字符缓冲区
 memset(&buff,0,1024);                               //初始化字符缓冲区
 recv(s,&buff,1024,NULL);                            //接收数据
}
void CData::~CData()                                 //析构函数
{
  }
```

上面的代码已经基本实现了自定义 CData 的基本方法。用户在使用该类时，必须在程序中包含该头文件 Data.h。否则，用户将不能正确使用自定义类 CData。

2. 使用CData类

用户在程序需要使用自定义类 CData 时，首先需要包含该类头文件。代码如下：

```
...                                         //省略部分代码
#include "Data.h"                           //包含自定义类头文件
```

🔔**注意：**在 VC 中包含头文件时，如果所包含的头文件是系统定义的头文件，则使用符号“<>”括起来。否则，使用符号“" "”括起来。

然后，再定义自定义类对象。代码如下：

```
...                                         //省略部分代码
CData dat;                                  //定义类对象
```

完成以上步骤后，用户便可以在程序中使用该对象调用相关的成员函数了。例如，当服务器端转发消息时，应该使用 CData 类中服务器发送函数对消息进行发送。代码如下：

```
void CQQDlg::Onsoc(WPARAM wParam,LPARAM lParam)
{
    switch (lParam)                         //判断套接字事件
{
      ...                                   //省略部分代码
    case FD_READ:                           //套接字读取事件
      {
        ...                                 //省略部分读取事件响应代码
     for(int i=0;i<5;i++)
{
  if(msg[i].qq.m_name1== m_list.GetItemText(i,3))
                                            //判断昵称是否存在
{
dat.send(msg[i].s,&msg[i].qq.msg,1024,NULL);
                                            //使用对象将信息转发到接收方
  }
}
```

用户使用自定义类 CData 中的成员时，仅需要使用该类对象对函数进行调用即可。所以，在本节中，不再对该类中其他的成员函数进行举例调用，请用户自行参考随书光盘中的实例代码。

11.3　Q 版邮件收发功能

在实例程序界面中，用户已经看到了 QQ 邮件按钮，因此用户可能会问为什么还没有实现该按钮的消息响应函数。在本节中，将主要向用户讲解该功能的具体实现方法以及回顾一下邮件的相关格式等。

11.3.1　信件格式和内容

在第 7 章中，已经向用户讲解了关于邮件发送与接收的相关知识。在本节中，将引导用户回顾一下前面所讲解的相关内容。首先，邮件的一般格式如下：

```
Data:Tue,04 Feb 2009 21:18:03+0800
From:lymlrl@163.com
Sender: lymlrl@126.com, wexs@163.com,wen@126.com,wuy@sina.com.cn
                                                               //发送者为多个地址
To:lymlrl@126.com,data@yahoo.com.cn,asj@sina.com.cn
                                                                //接收者也为多个
Subject: This is a E-mail                                       //邮件主题

Hello lymlrl!                                                   //邮件数据体
This is a E-Mail!
```

在上面的邮件格式中，Data 等字段分别描述了邮件的相关信息。例如，邮件发送日期、发送者以及接收者等。

然后，用户可以根据邮件的一般格式，构造一封简单的邮件。代码如下：

```
Data:Tue,04 Feb 2009 21:18:03+0800
From:lymlrl@163.com
Reply-to:lymlrl@sina.com.cn                                    //回复邮件地址
Sender: lymlrl@126.com, wexs@163.com,wen@126.com,wuy@sina.com.cn
To:lymlrl@126.com,data@yahoo.com.cn,asj@sina.com.cn
Subject: VC 编程!                                              //邮件主题

请教几个关于 VC 编程的问题?                                    //邮件体
  ...                                                          //省略部分邮件数据
```

一般邮件的格式如上所示，如果用户对邮件格式等不熟悉，请复习第 7 章中的相关内容。在本节中，不再赘述这些知识点。

11.3.2　邮件的基本语法

一般情况下，邮件是由格式加上语法构成。因此，在本节中，主要向用户讲解邮件语法方面的知识。首先，邮件数据是由一些 SMTP 邮件头字段标识，如表 11.4 所示。

表 11.4　SMTP邮件头字段

字　　段	意　　义	字　　段	意　　义
From	邮件创建者的邮件地址	In- Reply-To	邮件正被回复
To	邮件目的地	Data	邮件创建的时间
Sender	邮件发送者	Subject	邮件主题
Reply-to	邮件回复地址	Comments	邮件的其他说明
Cc	邮件抄送人	Bcc	邮件的密件抄送人邮件地址

在表 11.4 中，列举出了一些比较常用的 SMTP 邮件头字段，关于这些字段的含义以及用法请用户复习第 7 章相关知识点。例如，在邮件中标识邮件发送者和接收者的邮件地址。代码如下：

```
...                                 //省略部分代码
From:lymlrl@163.com                 //标识邮件发送者
To:lymlrl@126.com                   //标识邮件接收者
```

然后，请用户参照 11.3.1 节中邮件格式和本节中所讲解的 SMTP 邮件头字段，可以自行构造一封简单的邮件。

11.3.3　如何构造并发送一封邮件

首先，用户在构造一封邮件时，应该对该邮件的相关信息进行描述。代码如下：

```
Data:Tue,04 Feb 2009 21:18:03+0800
From:lymlrl@163.com
Reply-to:lymlrl@sina.com.cn                              //回复邮件地址
Sender: lymlrl@126.com, wexs@163.com,wen@126.com,wuy@sina.com.cn
To:lymlrl@126.com,data@yahoo.com.cn,asj@sina.com.cn
Subject: VC 编程!                                         //邮件主题
...                                                      //省略部分代码
```

在上面的信息描述中，主要是描述了邮件构造的日期、发送者、接收者以及邮件主题等。接下来，便可以构造邮件内容了。但是，需要用户注意邮件内容与邮件信息描述之间必须空一行，以区分两者内容。代码如下：

```
...                                                      //省略部分邮件描述

请教几个关于 VC 编程的问题?                                //邮件体
  ...                                                    //省略部分邮件数据
```

然后，邮件构造成功便可以将其发送到目的地址了。请用户参考第 7 章实例程序，本节不再向用户讲述发送邮件的相关编程方法了。

在本节实例程序中，用户可以通过单击“QQ 邮件”按钮调用第 7 章实例程序或者是 Windows 系统的邮件发送程序发送邮件。

如果用户采用第 7 章的实例程序发送邮件，则需要在“QQ 邮件”按钮的消息响应函数中，添加代码调用邮件发送实例程序。代码如下：

```
void CQQDlg::OnEmail()
{
    CString str="邮件收发器.exe";                          //定义文件路径
    STARTUPINFO si={sizeof(si)};                          //定义结构体变量
    PROCESS INFORMATION pi;
    CreateProcess(str.GetBuffer(1),NULL,NULL,NULL,true,NULL,NULL,NULL,&s
    i,&pi);
                                                         //调用进程打开邮件收发器
}
```

除了使用上述使用的邮件收发器之外，用户也可以使用 Windows 系统的邮件发送程序进行邮件服务操作。代码如下：

```
void CQQDlg::OnEmail()
{
  ::ShellExecute(NULL,NULL,"mailto:lymlrl@163.com",NULL,NULL,SW SHOW);
}
```

与前面的方法相比，调用 Windows 系统的邮件发送程序对邮件进行发送比较简单。但是，用户使用这种方法具有很大的局限性。因为用户在程序中只能指定一个固定的邮件接收者的 E-mali 地址。当然，为了消除这样的限制，用户也可以通过一个简单的对话框，设

置邮件接收者的 E-mali 地址。

实现由用户设置邮件接收者，必须在实例工程中添加一个对话框，并将该对话框关联一个新类 CSet。然后，在窗口类中声明该对话框类的实例对象。代码如下：

```
#include "Set.h"
class CQQDlg : public CDialog
{
public:
        CSet set;                       //定义邮件地址设置对话框对象
...                                     //省略部分代码
}
```

当窗口对象定义以后，用户可以在 QQ 邮件按钮的消息响应函数中编程实现弹出邮件地址设置对话框。代码如下：

```
void CQQDlg::OnEmail()
{
set.DoModal( );
}
```

保存、编译并运行程序，当用户单击“QQ 邮件”按钮后，程序会弹出邮件地址设置对话框，如图 11.30 所示。



图 11.30　邮件地址设置对话框

用户在弹出的邮件地址设置对话框中输入邮件接收者的 E-mail 地址，单击 OK 按钮。接着程序会调用 Windows 系统的邮件发送程序，代码如下：

```
void CSet::OnOK()                                   //OK 按钮消息响应函数
{
CString mail,str;                                   //定义字符串变量
mail+="mailto:";                                    //连接字符串
GetDlgItem(IDC_EDIT1)->GetWindowText(str);          //获取用户输入的邮件地址
if(str.GetLength()==0)                              //判断输入是否为空
{
    MessageBox("邮件接收地址不能为空!");             //弹出消息框
}
else
{
mail+=str;                                          //连接邮件地址字符串
```

```
::ShellExecute(NULL,NULL,mail,NULL,NULL,SW_SHOW);
                                                        //调用邮件发送程序
::SendMessage(this->m_hWnd,WM_CLOSE,0,0);               //关闭设置对话框
}
}
```

上面的代码中，用户调用邮件发送程序成功之后，必须发送关闭消息关闭地址设置对话框。否则，程序界面会显得不友好。该程序运行效果请用户参考随书光盘中的实例代码。

在本节中，主要是将本实例与第 7 章邮件收发器的相关知识结合起来向用户讲解实现邮件发送的功能。具体代码与运行效果请用户参考随书光盘。

11.4　Q 版浏览器

Q 版浏览器是指用户在本章实例中，可以通过界面中的按钮打开浏览器浏览相关网页。在第 5 章中，已经向用户讲解了 VC 实现网页浏览器功能的方法。所以，本节中仅向用户介绍 URL 编码的相关知识以及在仿 QQ 实例中使用浏览器浏览网页。

11.4.1　URL 编码

URL 编码是一种浏览器用来打包表单输入的格式。一般情况下，浏览器从表单中获取所有的变量以及其中的值，并移去或者替换那些不能进行传送的字符等。当然，程序在进行 URL 编码时，还取决于用户所使用的数据传送方式是 GET 还是 POST 方式。

用户在浏览网页时，经常会遇到一些向服务器提交数据等操作。此时，用户仔细看看网页的网址会发现网址后面连接了很长的一段字符。这些字符便是用户提交的数据，只是该数据已经被进行了 URL 编码。所以，用户看不懂这串长长的字符。例如，当用户登录邮箱时，浏览器地址栏中显示的提交网页如下：

```
https://reg.163.com/logins.jsp?type=1&url=http://entry.mail.163.com/
coremail/fcg/ntesdoor2?lightweight%3D1%26verifycookie%3D1%26language%3D
-1%26style%3D21
```

在上面的网址中，语句“type=1”表示登录方式为验证登录。而后面紧跟着的一串字符“%3D1%26verifycookie%3D1%26language%3D-1%26style%3D21”，则表示用户输入的用户名以及密码已经经过了 URL 编码。当服务器接收到用户的请求后，其脚本程序会从该网址中获取用户的用户名以及密码并与数据库中的相应数据进行比较，判断用户是否登录成功。

由于在本实例中，仅需要用户调用浏览器实现浏览网页的功能。所以，关于 URL 编码方面的知识，本书不再进行介绍。如果用户对此感兴趣，请参考其他一些专门讲解网页浏览器相关知识的书籍。

11.4.2　使用浏览器

用户在仿 QQ 实例中，可以使用两种方法调用网页浏览器浏览网页，分别是使用第 5

章中设计的实例程序和使用 Windows IE 浏览网页。由于在程序中，调用前者实现浏览网页的方法与前一节中调用邮件收发器的方法相同。所以在本节中，主要向用户讲解第二种实现方法。

（1）用户在实例工程中添加一个用于指定网页地址的对话框，并为其关联新类名 CNet。然后，再在实例对话框类中定义 CNet 类的实例对象。代码如下：

```
#include "Net.h"                              //包含头文件
class CQQDlg : public CDialog
{
public:
        CNet net;                             //定义网页地址设置对话框对象
...                                           //省略部分代码
}
```

（2）用户可以在“QQ 网页”按钮的消息响应函数中，弹出网页地址对话框。代码如下：

```
void CQQDlg::OnNet()
{
   net.DoModal( );                            //弹出网页地址设置对话框
}
```

（3）保存、编译并运行该程序，当单击“QQ 网页”按钮时，程序会弹出网页地址设置对话框，如图 11.31 所示。



图 11.31　弹出网页地址设置对话框

用户在该对话框中输入网页地址后，单击 OK 按钮便可以打开 IE 浏览浏览相关的网页了。代码如下：

```
void CNet::OnOK()                                          //“OK”按钮消息响应函数
{
CString netadd,str;                                        //定义字符串变量
GetDlgItem(IDC_EDIT1)->GetWindowText(netadd);              //获取用户输入的网页地址
if(netadd.GetLength()==0)                                  //判断输入是否为空
{
    MessageBox("网页地址不能为空!");                        //弹出消息框
}
```

```
else
{
    if(netadd.Find("www")!=-1)                          //判断用户的输入是否正确
    {
    ShellExecute(NULL,"open",netadd,NULL,NULL,SW_SHOW);
                                                        //调用 IE 浏览器浏览网页
    }
    else
    {
        str+="www.";                                    //添加字符串
        str+=netadd;
        ShellExecute(NULL,"open",str,NULL,NULL,SW_SHOW);
    }
::SendMessage(this->m_hWnd,WM_CLOSE,0,0);               //关闭该对话框
}
}
```

在代码中，用户首先判断输入是否为空。若不为空，则验证输入的网址是否符合规范。否则，提示用户网页地址不能为空。网址验证成功之后，调用 IE 浏览器浏览相应的网页。

11.5　小　　结

在本章中，向用户介绍了仿 QQ 软件的制作过程，以及各个功能的实现方法。在具体的编程过程中，不仅介绍了实例程序的工作原理等，还向用户讲解了界面的设计、控件的使用、发送邮件以及打开网页等功能的具体实现方法。通过本章实例的实现方法，用户可以学习到关于程序界面设计等方法的应用。

第 3 篇　Visual C++ 串口通信

第 12 章　串口通信基础

在日常生活中，计算机串口对于用户而言，有着非常广泛的用途。例如，工业控制、计算机串口通信等。因此，串口通信编程是实现这些用途的最好途径。在本章中，将向用户介绍串口通信编程的基础知识以及串口通信数据的校验方法等。

12.1　串口通信基本概念

用户需要进行串口编程，必须对串口通信的一些基本概念以及通信数据传输的方式等非常地熟悉。因此，在本节中，主要向用户介绍一些关于串口通信方面的基础知识。

12.1.1　串口通信概述

串口通信是指用户通过计算机串口实现计算机与计算机之间的通信。一般情况下，串口均是按位（bit）进行发送和接收数据。所以，计算机串口常被用于远距离传输信号或者数据。串口通信编程中，最重要的参数包括波特率、数据位、停止位等。当两台计算机通过串口进行通信时，必须将这些参数设置为相同。否则，两台计算机将不能进行数据通信。

1．波特率

波特率是指用户每秒钟通过串口进行数据传输的位个数。波特率在计算机串口通信中，是一个非常重要的参数，常被用于衡量通信的速度。例如，用户在进行串口编程时，将波特率设置为 9600，其表示的意思是串口每秒钟传输的数据个数为 9600 B。

注意： *用户在使用串口进行通信时，波特率可以为任何值。但是，用户在设置波特率时，应该综合分析之后再进行设置。默认情况下，波特率为 9600。*

波特率也可以被用来描述串口通信的距离。一般情况下，波特率越大，其数据传输距离越短。如果用户需要进行远距离数据传输时，需要将波特率设置得越小。

2．数据位

数据位是指在计算机串口通信中，用来描述实际传输数据位的参数。其中，实际传输的数据位包括开始位、停止位、数据位以及奇偶校验位。这些数据位均包含在一个数据包中进行传输。

3．停止位

停止位是指计算机发送或接收的每个数据包的最后一位。因为计算机通信数据都是在

传输电缆中进行传输的，所以，计算机发送的数据会受到计算机时钟的影响，导致停止位不能简单的被用于表示数据传输的结束。

如果用户设置的停止位位数越多，则其数据传输的速率会越慢。这是由于计算机串口数据传输的特点是按位进行传输的。

注意：用户在实际编程时，为了避免停止位与用户所传输的数据位相同，造成数据传输的混乱。所以，用户需要将串口数据的停止位的位数增多。

4．奇偶校验位

奇偶校验位在串口通信中，是一种最简单的检错方式。其中分别包含了四种校验方法：奇位校验和偶位校验。当然，在串口通信中，没有校验位也是允许的。

用户在进行串口编程时，必须设置至少一个检验位，以确保用户传输的数据的完整性和准确性。当用户传输的数据中“1”的个数为偶数时，则校验位为“1”。否则，校验位为“0”。当接收方接收到数据时，将首先按照校验位，对数据中的“1”的个数进行检测。如果为奇数，则表示传送正确。否则，表示传送错误。

注意：偶校验和奇校验的基本原理是相同的，只需检测数据中的“1”或“0”的个数值是偶数还是奇数。

12.1.2　单工、半双工和全双工的定义

一般，根据串口数据的传输方向，可以将串口通信方式大致分为单工、半双工和全双工。用户在使用串口进行编程时，必须需要知道串口的通信方式。所以，在本节中，将向用户分别介绍单工、半双工和全双工的基本定义。

1．单工

单工是指在串口通信中，其通信数据只能由一端向另一端进行单向传输。一般，串口单工通信方式常被应用在工业控制方面。例如，工业计算机通过串口，从传感器中获取采样数据等。具体的单工通信方式如图 12.1 所示。

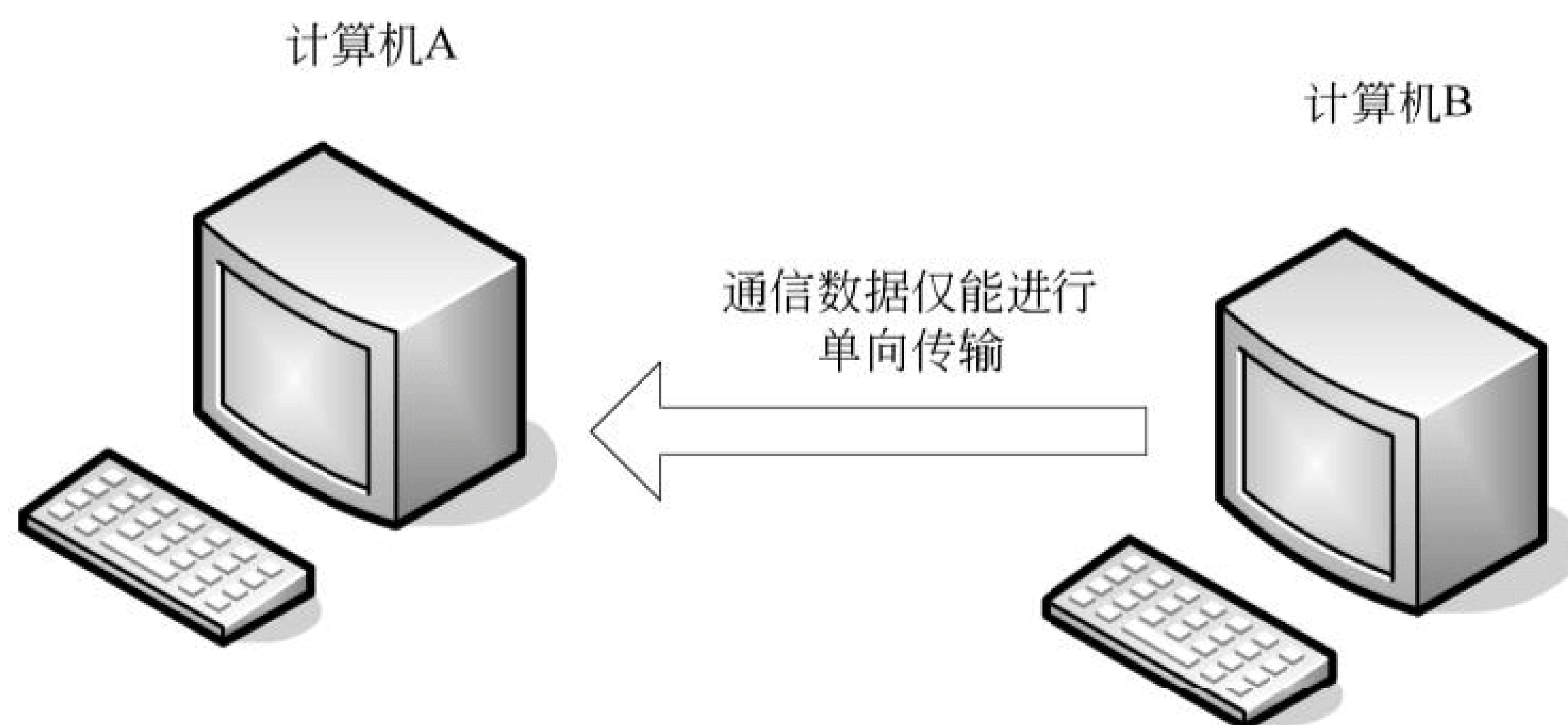


图 12.1　串口通信的单工通信方式

🔔**注意：** 如果串口采用单工通信方式进行通信，其通信数据仅能从计算机 B 到计算机 A。数据传输方向不能逆转。目前，这种通信方式已经很少应用在实际项目的开发中。

2. 半双工

半双工通信是指串口数据可以从通信的一端传输到另一端，而该数据传输方向也可以进行逆转。但是，计算机在串口通信的半双工方式下，并不能同时发送和接收数据。所以，用户采用半双工方式进行串口通信时，只能允许一个方向上的数据传输。半双工通信方式如图 12.2 所示。

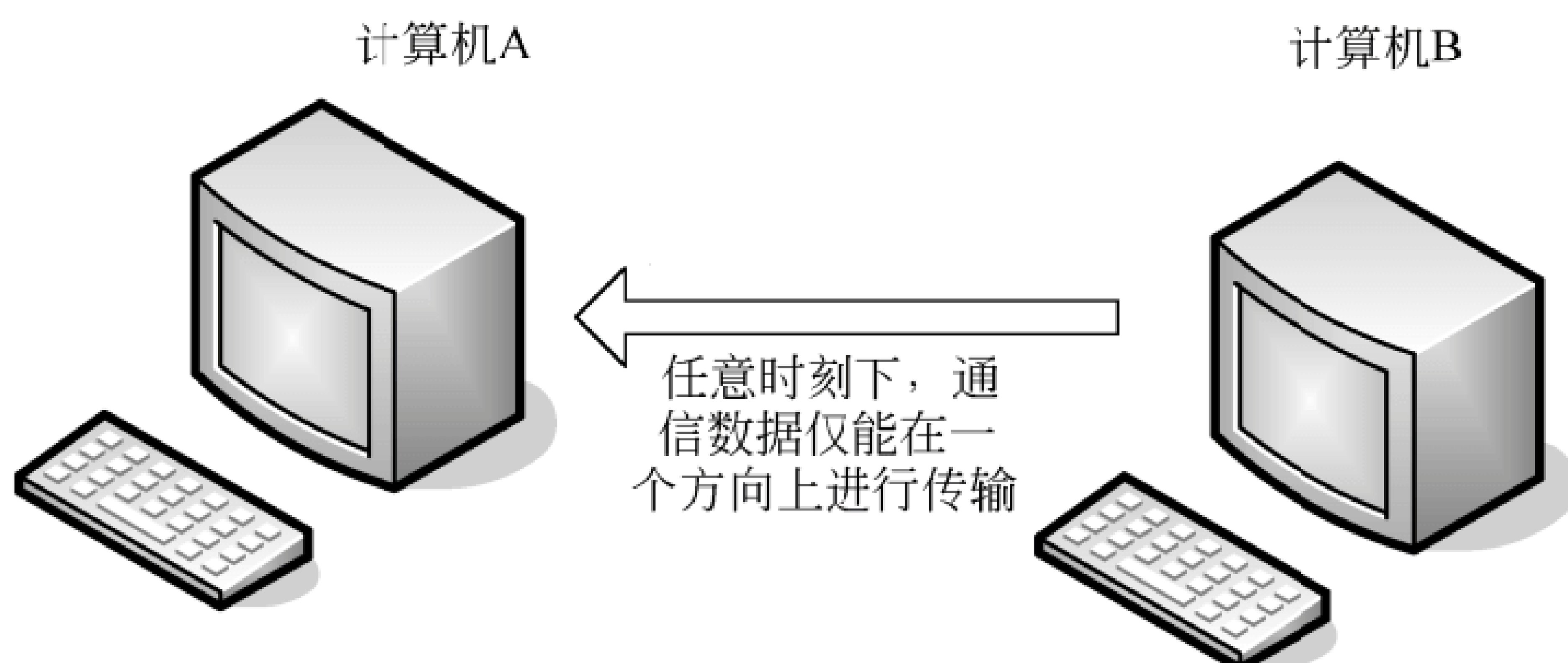


图 12.2　串口通信的半双工通信方式

🔔**注意：** 在串口通信中，采用半双工方式进行数据传输时，用户需要使用到两根数据传输线。但是，在任意时刻，这两根数据传输线只能允许其中一根存在数据传输。

3. 全双工

全双工是指在任意时刻下，串口通信数据可同时在传输线路上，进行双向传输。使用该通信方式时，用户需要使用两根数据传输线，一根数据传输线发送数据，而另一根数据传输线则接收数据。这种通信方式已经被广泛使用到实际的项目开发中。全双工通信方式如图 12.3 所示。

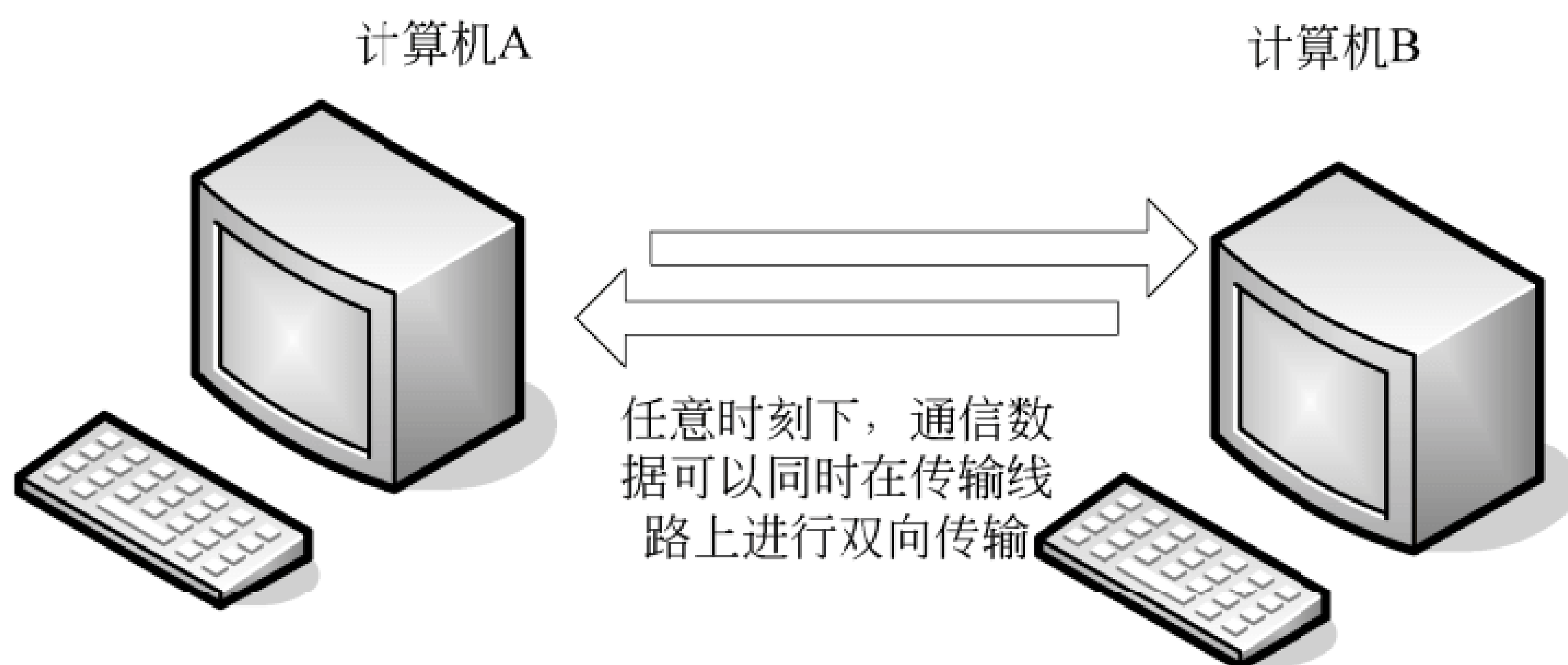


图 12.3　串口通信的全双工通信方式

在本节中，向用户介绍了单工、半双工以及全双工串口通信方式的基本工作原理。用户在进行串口编程时，一定需要首先规定串口的通信方式。

12.1.3　同步方式与异步方式

在串口通信中，除了 12.1.2 节向用户介绍的单工等通信方式以外，串口的通信方式还可以分为同步方式和异步方式。本节将向用户介绍这两种通信方式的基本原理以及区别。

1．同步方式

同步通信方式是指在串口通信编程中，用户从串口读取或者写入数据时，其线程函数会发生阻塞。当用户使用同步方式传输数据时，程序会在该操作上进行等待，直到该操作有返回值返回为止。

用户使用同步方式传输数据时，是将数据一个一个地进行传输。但是，在同步方式下，不允许传输的数据之间存在空位。所以，数据在进行同步传输前，必须填充空数据位。一般，进行同步传输时，均以同步字符作为数据的开始。如果接收方接收到该同步字符，则将其之后的数据认为是实际传输的数据进行处理。

串口通信的同步方式按照同步字符的不同，可以分为面向字符、面向比特等同步方式。首先，面向字符的同步方式是按照一定的格式进行数据传输，该格式如表 12.1 所示。

表 12.1　面向字符的同步方式数据格式

SYN	SOH	标题	STX	数据块	ETB/ETX	块校验

用户通过表 12.1 所示的数据格式，可以看到在同步方式下传输数据所使用的全部控制字符，这些控制字符的意义如表 12.2 所示。

表 12.2　串口同步控制字符意义

控 制 字 符	意　　义	控 制 字 符	意　　义
SYN	同步字符	数据块	实际传输的数据
SOH	开始标题标识	ETB	标识数据块传输结束
标题	包含发送方地址以及接收方地址	ETX	标识全部数据传输结束（包括多个数据块）
STX	实际传输数据标识	块校验	整个数据的校验码

面向字符的同步方式，最大的缺点在于当数据发送时，如果实际数据与同步字符相同，则接收方将无法识别数据的完整性和准确性。

如果用户采用面向比特的同步方式进行数据传输，则需要使用特定的八位二进制数作为传输数据的开始或者结束标志。其数据格式如表 12.3 所示。

表 12.3　面向比特的同步传输方式数据格式

01111110	A	B	C	D	01111110

在该数据格式中，是以二进制数 01111110 作为数据的开始和结束标志。其中，数据格式中，各个字节的含义如下：

- A 表示接收方的地址字节。当接收方接收到数据后，会检查这个地址，若地址字节的第一位为 0，则表示其后面是一个地址字节。若为 1，则表示该字节后面是最后一个地址字节。

注意：地址字节的位数必须是 8 的整数倍。

- B 表示控制字节。表示传输数据的类型。若控制字节的第 1 位为 0，则表示该字节后还有一个字节，并且这个字节也是控制字节。
- C 表示实际传输的数据。
- D 表示循环冗余校验位。

在本小节中，主要向用户介绍了同步传输方式的基本原理以及数据格式等。通过本小节的学习，用户应该可以学会构造用于进行同步传输的串口数据。

2. 异步方式

异步传输方式与同步传输方式恰好相反。异步传输方式是指程序可以将传输数据的处理交给一个线程或者进程完成，而程序本身则可以进行其他数据的处理。该传输方式是一种非阻塞方式。用户在实际编程时，可以将其视为一种多线程工作方式。

采用异步传输方式传输数据的发送方可以在任意时刻将数据发出，而接收方也可以在任意时刻，接收数据。因此，在串口通信时，采用异步传输方式可以提高程序的运行效率。

注意：异步传输方式是以字符为单位进行数据传输的。

12.1.4　串口通信的应用方向

目前，由于串口能进行远距离数据传输。所以，串口通信的应用方向十分广泛，常被用作工业控制、工业通信、数据传输等。

通过串口，计算机可以实现控制一台或多台下位机，实现计算机控制自动化。这样，用户不但可以节约成本，还可以最大限度的发挥计算机的作用。

在科技日益发达的当今时代，计算机串口会越来越多地被应用到各个行业中。因此，用户学习计算机串口编程将显得尤为重要。

12.2　常用数据校验法

在 12.1 节中，已经向用户介绍了一些串口编程的基础知识。本节将主要向用户介绍在串口通信中，最为常用的两种数据校验方法。这两种校验方法分别是奇偶校验以及循环冗余校验。

12.2.1　奇偶校验

在串口通信中，其通信数据会受到外部干扰，导致数据的完整性和准确性遭到破坏。

因此，用户为了使通信数据完整、准确地到达接收方，需要使用一些校验数据的方法。其中，最为简单的一种方法便是奇偶校验法，但是该方法仅能检错，而不能纠错。所以，当用户检查到错误时，只有要求发送方重新发送数据。一般，在奇偶校验法中，包含了奇校验以及偶校验两种。

1. 奇校验

奇校验法是指在通信数据中，数据位 1 的个数应该为奇数个。此时，通信数据的校验位为 1，否则为 0。当接收方接收到数据后，将各个数据位相加。若相加后和为奇数，则表示通信数据完整而且正确。否则，通信数据出现错误，接收方需要向数据发送方请求数据重发。

例如，用户定义通信数据为“01011110”，其中最后一位为奇校验位。用户将这个数据相加后，其和为奇数 5。但是，其校验位却为 0，表示该数据在传输过程中，受到了外界的干扰。此时，由于奇偶校验法只能检错而不能进行纠错，所以接收方只能要求发送方重新发送该数据。

2. 偶校验

偶校验法是指在通信数据中，数据位 1 的个数应该为偶数个。此时，通信数据的校验位应该设置为 0，否则需要设置为 1。当接收方接收到该数据后，同样是将各个数据位相加。若相加后其和为偶数，则表示通信数据完整而且正确。否则，该通信数据出现错误，接收方需要向数据发送方请求数据重发。

例如，用户定义一个通信数据为“01101100”，其中最后一位是偶校验位。此时，用户将该数据的各个数据位进行相加，其和为偶数 4。由于该通信数据的校验位为 0，所以，接收方接收到的数据完整而正确。

3. 奇校验和偶校验的区别

在串口数据校验中，奇校验主要用于判断数据中 1 或 0 的个数是否为奇数。如果该数据的校验数据位上的数据为 1，那么表示该数据在传输过程中，未受到外界的干扰。否则，当校验数据位上的数据为 0 时，表示该数据的完整性和准确性已经受到外界干扰。此时，接收方必须要求发送方重新发送数据。

而偶校验主要用于判断数据中的 1 或 0 的个数是否为偶数。当该数据的校验数据位上的数据为 0 时，表示该数据在传输过程中，未受到外界的干扰。否则，当校验数据位为 1 时，表示数据在传输过程中，受到了外界的干扰。

不论是奇校验法，还是偶检验法，其基本实现原理都是一样的。因此，用户在实际使用时，需要指定数据中的特征数据位是 1 还是 0，按照其数据位个数设置相应的校验数据位即可。

注意： 奇偶校验法只能检错，而不能实现纠错功能。

12.2.2　循环冗余校验

一般情况下，在使用串口通信时，为了能够对数据进行检错并纠错，用户都会在数据

校验中广泛采用循环冗余校验这一方法。在本节中，将向用户介绍在串口通信中常用的循环校验法。

循环冗余校验码是由两部分组成，前一部分是信息码，也就是需要校验的信息。而后一部分则是校验码。如果循环冗余校验码的总长度为 n，而信息码长度为 k，则可以将其称为（n,k）码。一般情况下，其编码规则均由几个步骤构成。

（1）将通信数据中的信息码左移 m 位（k+m=n）。例如，通信数据为“01101101”，接着，用户将该数据向左移动 3 位。移动后的数据为“01101000”。

（2）将移动后的数据与一个用于生成校验码的二进制数相异或，所得到的二进制数便是校验码。

例如，用户定义的用于生成校验码的二进制数为“111101”，现在将该二进制数与移位后的数据进行异或操作。具体计算方法是，将数据的前 6 位与自定义的校验码相异或。计算式如：111101 ^ 011011=100110。再将刚才异或时，通信数据锁剩下的最后两位（即“01”）与该结果进行异或的结果为“100111”。最后，用户再将该结果与校验码“111101”进行异或，得到最后的结果是“011010”。用户最终得到的结果也就是所需要的校验码。

注意： 当数据的接收方接收到数据时，必须使用上面所使用的用于生成校验码的二进制数为“111101”与接收数据进行运算，恢复数据的原样。在本节中，所介绍的异或运算是指相同的二进制数异或得到 0，而不同的二进制数异或得到 1。其运算的本质等同于加法运算，而不同的地方是相加而不进位。

12.3 小　　结

在本章中，主要从串口通信编程的角度向用户介绍了串口通信编程中将使用或者遇到的一些基本概念，并且使用图例向用户重点讲解了串口通信的几种模式。用户从这些通信模式中可以清楚地看到通过串口进行通信的基本过程。

本章还向用户讲解了通过串口进行数据传输时，一些常用的数据校验方法。在文中，较详细地举例说明每种方法的基本原理等。在第 13 章中，将向用户讲解实现串口通信程序设计实例的过程。

第 13 章　串口通信编程应用

如今，越来越多的用户将计算机串口应用到实际生产和生活中。利用串口进行数据通信，不但可以实现远距离数据传输，还可以轻松实现数据的检错与纠错。用户在 VC 中，实现串口通信编程可以使用 MFC 中的串口控件以及 Windows API 函数。所以，在本章中，将向用户分别介绍这两种实现方法。

13.1　MFC 串口控件编程

在 MFC 类库中，用户可以使用串口控件实现串口编程。该控件相当于用户自定义的类，而其中的每个功能都是由该类中的成员函数实现的。在 VC 开发环境中，用户可以在项目中插入串口控件，然后再为其关联一个类名即可。本节将主要向用户介绍如何使用 MFC 串口控件实现串口通信编程。

13.1.1　VC 中应用 MSComm 控件编程步骤

用户在 VC 开发环境中，使用 MSComm（串口）控件进行编程，必须首先将该控件添加到用户的工程项目中。因此，本节主要向用户讲解在已创建的实例工程中，插入串口控件的基本步骤。

1．创建工程

在 VC 中，用户需要创建基于 MFC 串口控件编程的实例工程。但是，用户在设置该工程的相关信息时，必须为该工程指定包含 ActiveX 控件的功能。否则，用户所创建的工程将不能使用 MFC 串口控件。其创建步骤如下：

（1）选择“文件”|“新建”命令，打开“新建”对话框，如图 13.1 所示。用户在新建工程对话框中，修改该工程实例的名称为“使用 MFC 串口控件实现串口编程”。

注意： 在这一步，用户必须选择工程的类型为 MFC AppWizard[exe]。否则，用户所创建的工程将不能使用 MFC 串口控件。

（2）用户修改完工程名称后，单击“确定”按钮，开始设置该工程的相关信息，如图 13.2 所示。由于该串口实例工程是基于单文档的，所以，用户需要在这一步将应用程序的类型指定为单文档。

（3）单击“下一步”按钮，选择是否包含数据库，如图 13.3 所示。

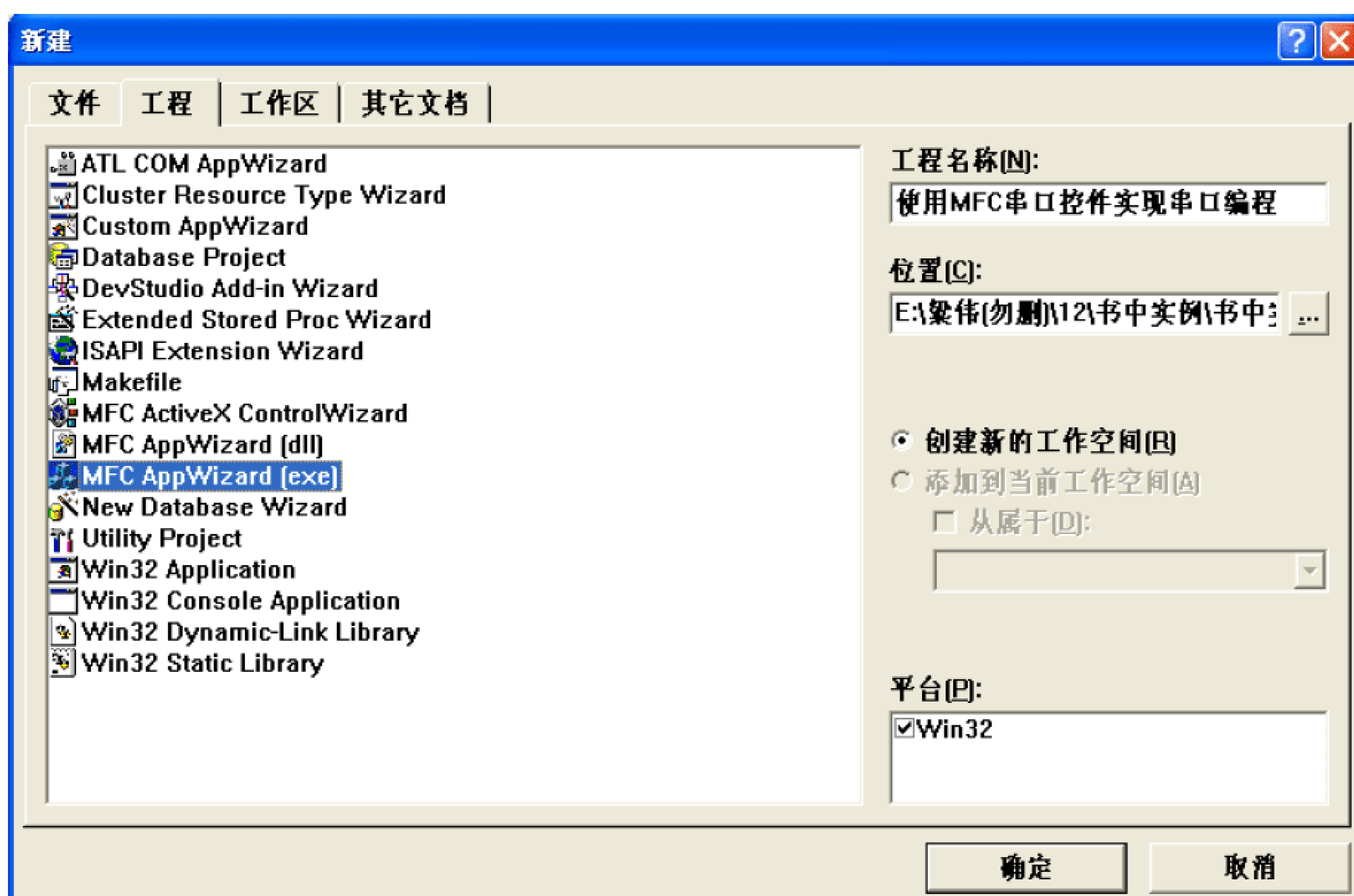


图 13.1　新建工程对话框

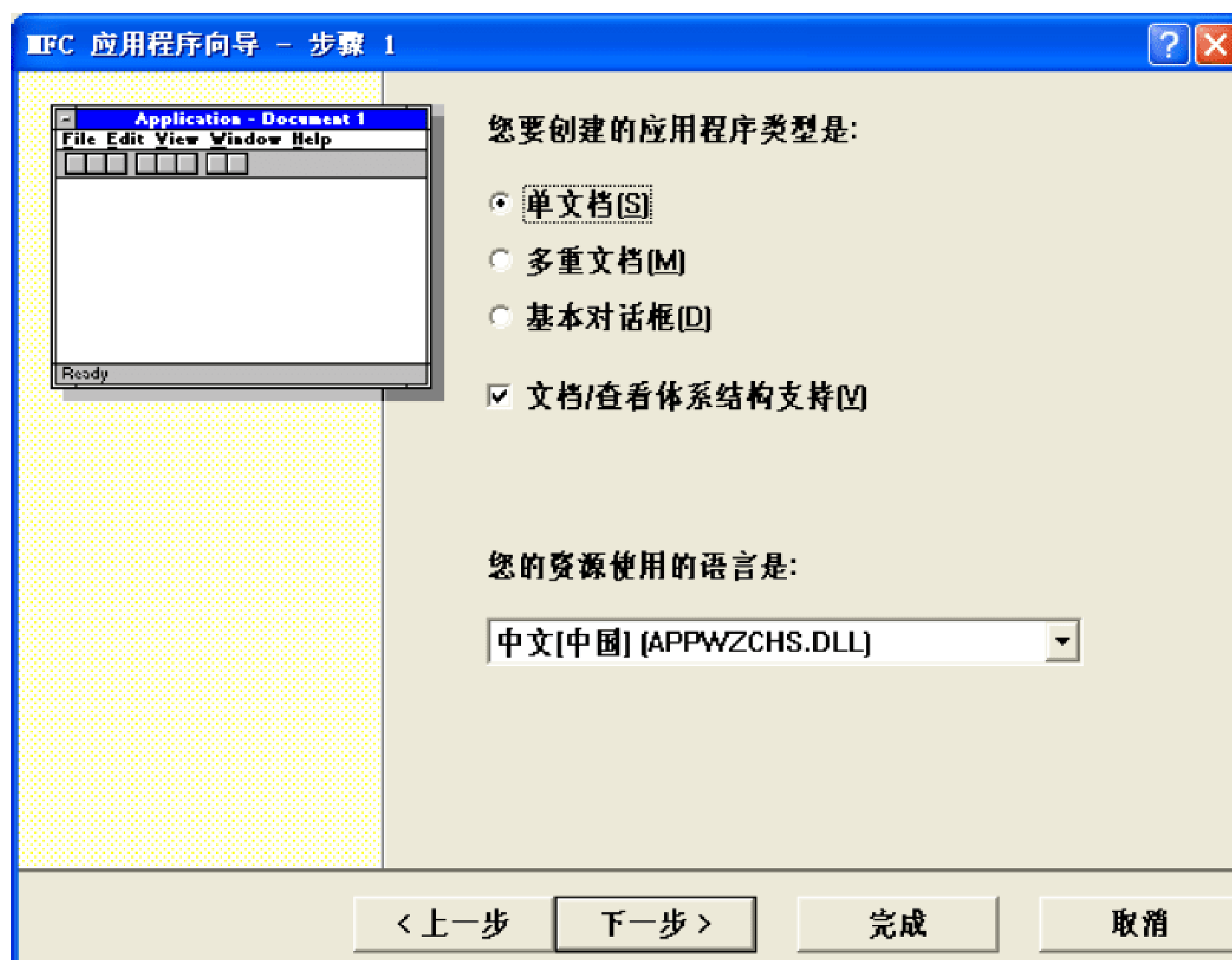


图 13.2　选择应用程序类型

注意：如图 13.3 所示，用户在该工程中不需要任何的数据支持。所以，在向导设置的该步骤中，选择“否”单选按钮即可。

（4）单击“下一步”按钮，为该工程选择支持 ActiveX 控件，如图 13.4 所示。用户为了在该工程中使用串口控件，必须在这一步指定其包含 ActiveX 控件的支持。

（5）单击“下一步”按钮，直接跳到设置步骤的第 5 步，设置应用程序的风格，如图 13.5 所示。在本章实例中，是将该应用程序的风格设置为默认风格。

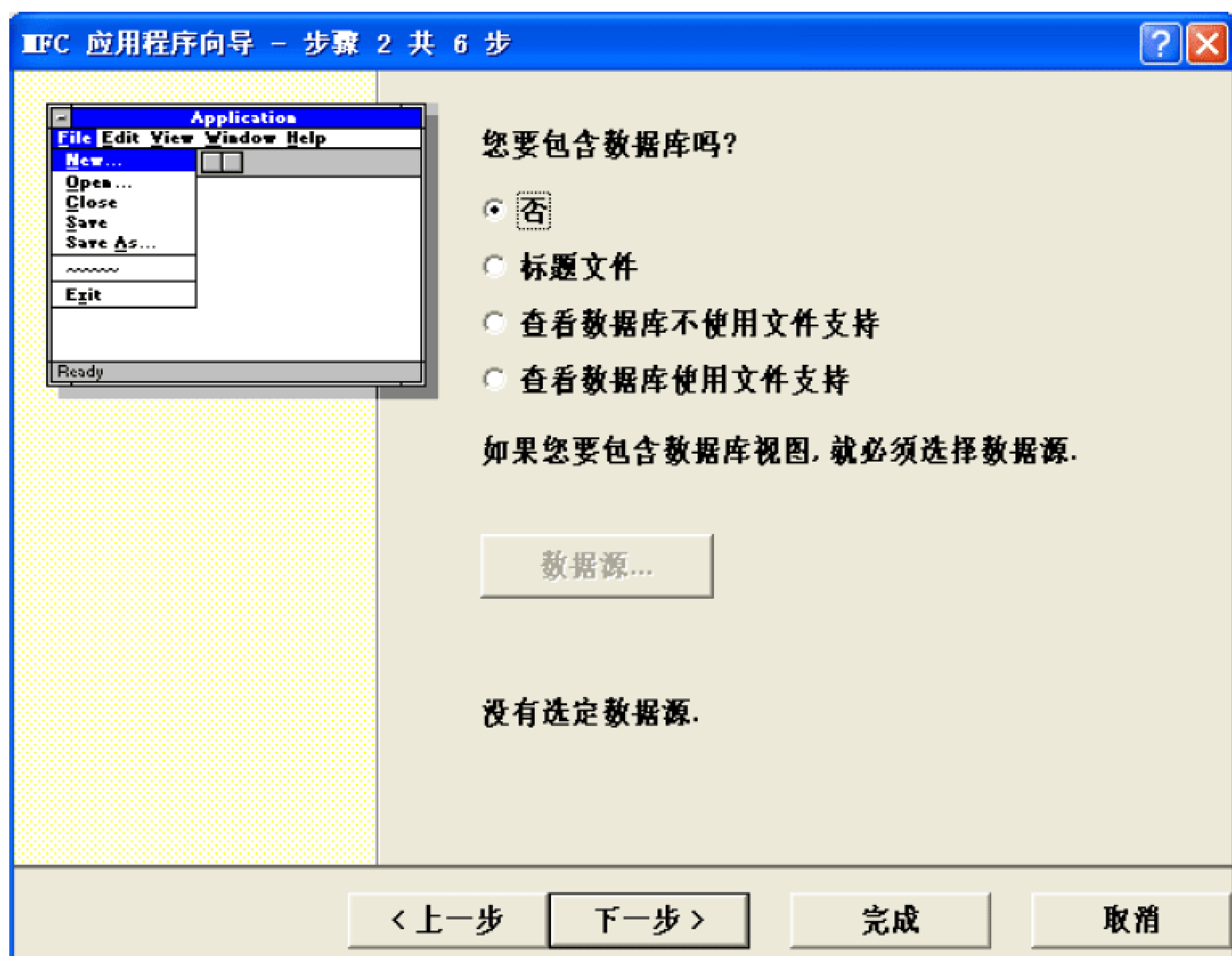


图 13.3　选择是否包含数据库

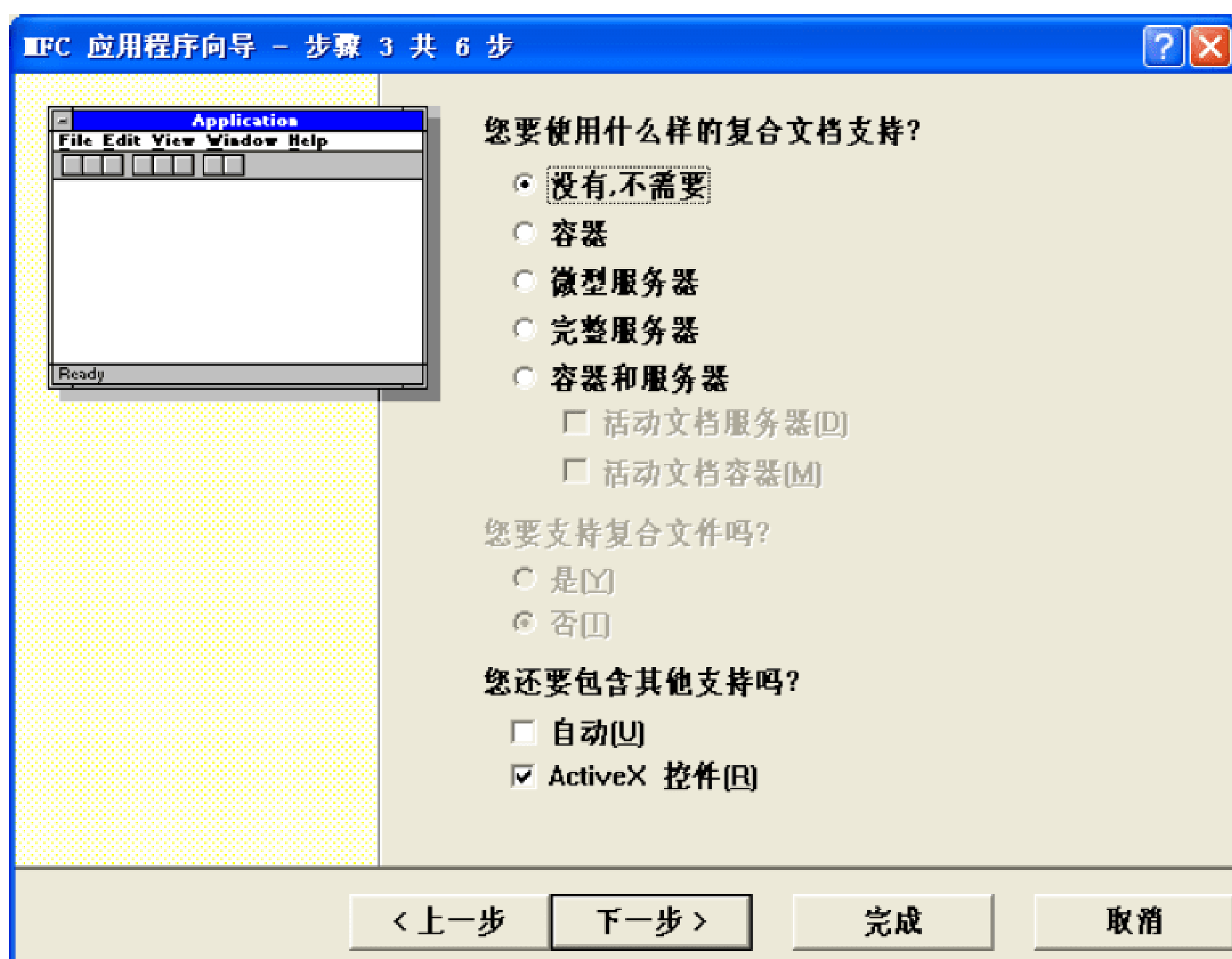


图 13.4　选择支持 ActiveX 控件

（6）单击“下一步”按钮，指定文档视图类的基类，如图 13.6 所示。

用户修改工程视图类的基类以后，直接单击“完成”按钮，完成工程的设置。

注意： *在该实例工程中，用户将该工程视图类的基类设置为 CFormView。这是由于考虑到用户在实际编程时，这样做会减少不必要的程序代码。*

2．向工程中添加串口控件

用户在工程名为“使用 MFC 串口控件实现串口编程”的实例工程中，可以通过菜单

插入串口控件。

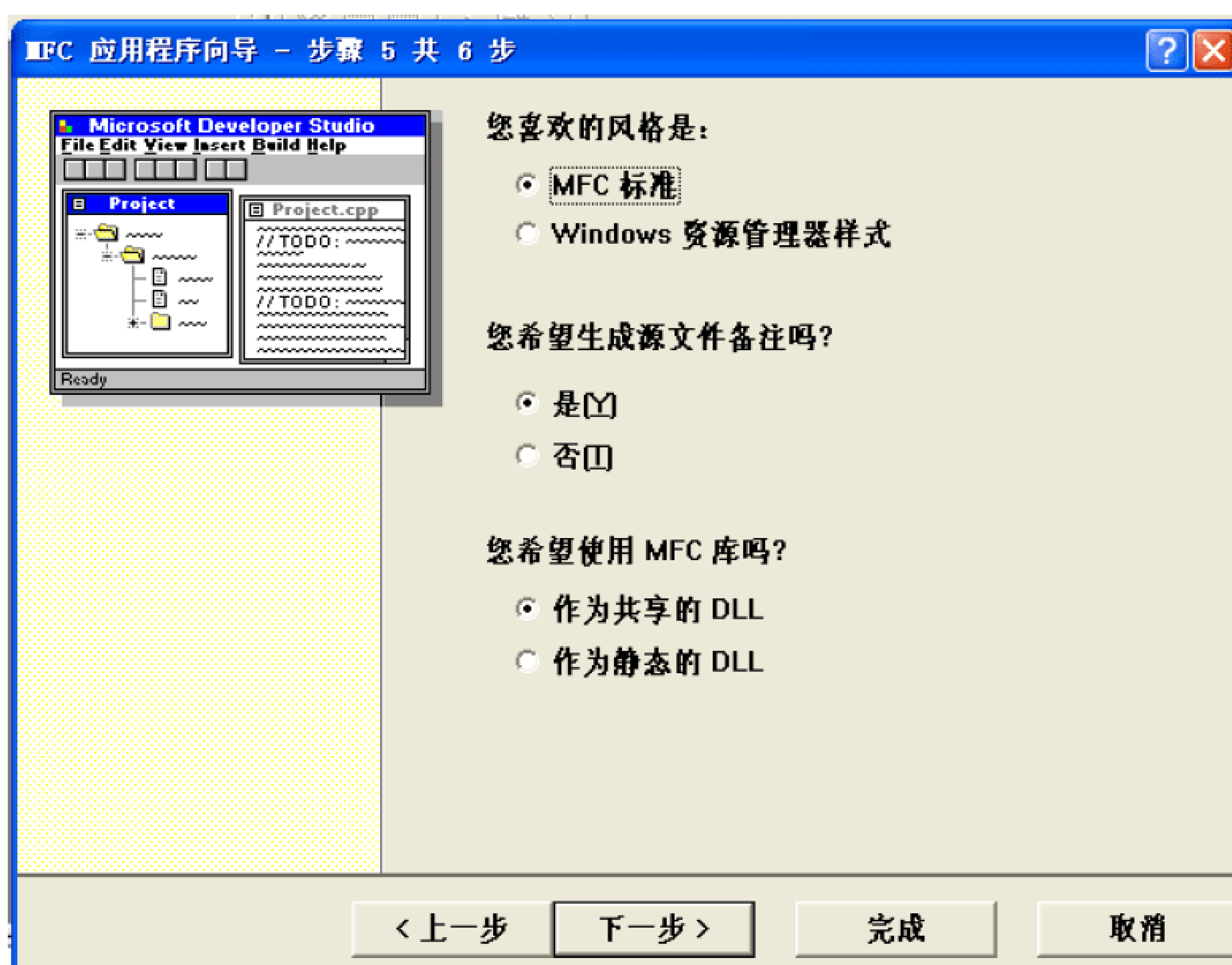


图 13.5　设置应用程序的风格



图 13.6　指定视图类的基类名

首先，选择“工程”|“增加到工程”|“Components and Controls Gallery”命令，打开 Components and Controls Gallery 对话框，如图 13.7 所示。

在用户的计算机没有注册 MSCOMM32.OCX 控件的情况下，用户是不能使用该控件的。此时，用户只能通过运行命令“regsvr32+控件的完整路径名”完成控件的注册，如图 13.8 所示。

如果该命令执行成功，则会弹出控件注册成功对话框，如图 13.9 所示。

通过以上步骤，用户已经在项目工程中，成功地添加了 MFC 串口控件。接下来，用户便可以为该控件关联串口控件类 CMSComm，并使用该类中的成员函数完成串口通信功能。

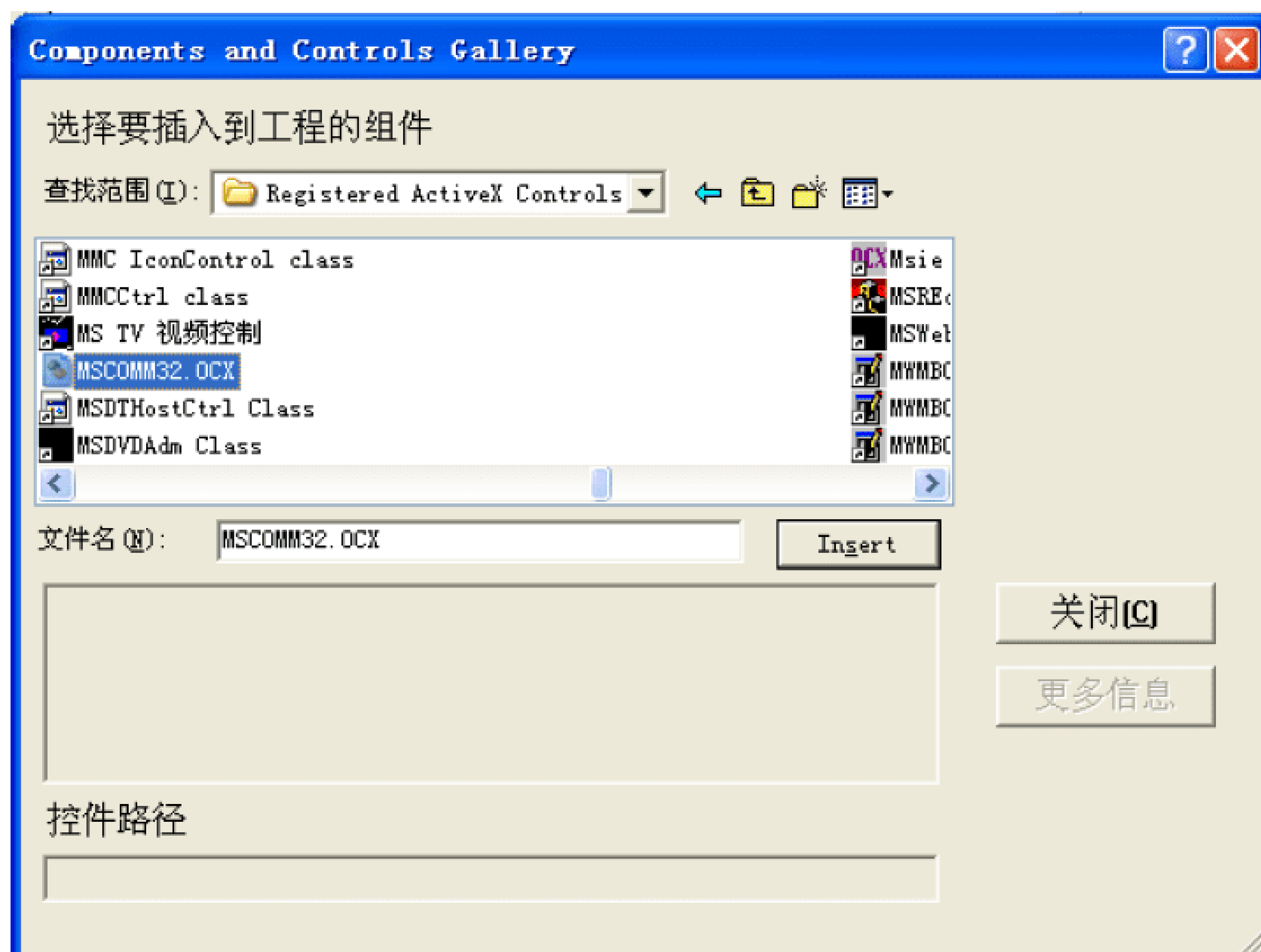


图 13.7　添加串口控件

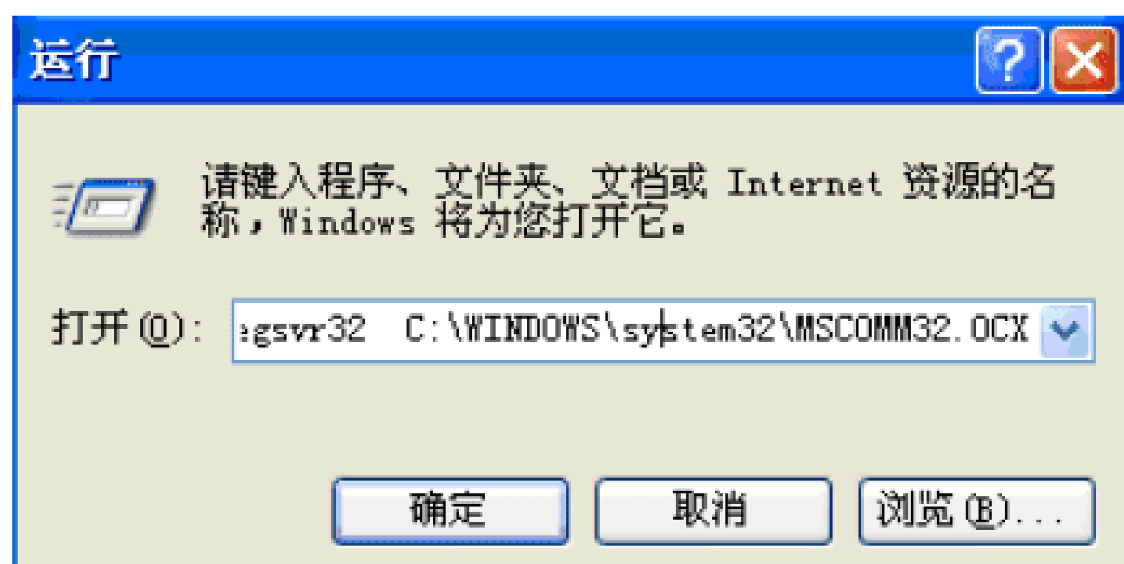


图 13.8　注册串口控件

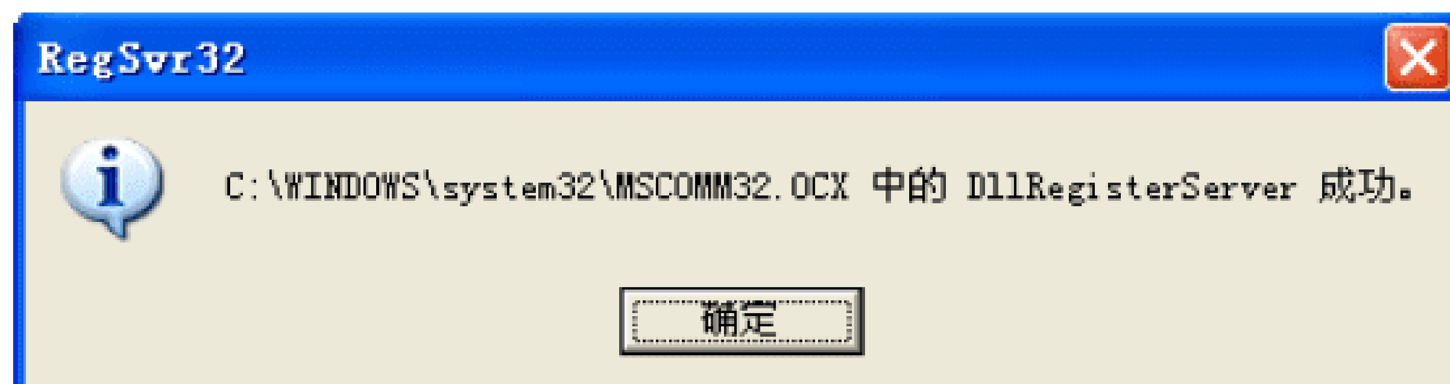


图 13.9　提示控件注册成功

13.1.2　MSComm 控件类

在 MFC 中，串口控件的类名为 CMSComm。用户在程序中可以使用该类的构造函数，创建该类的实例对象。然后，使用该对象调用其成员函数可以实现串口的相关功能。有关串口类成员函数的讲解将在 13.1.3 节进行。本节将主要向用户讲解 CMSComm 类的相关定义以及使用该类的相关方法。

1．CMSComm类头文件

用户在使用 CMSComm 类前，需要对该类中的相关变量以及成员函数进行比较细致的了解。所以，在本节中，将专门向用户介绍 CMSComm 类的头文件的具体定义。该头文件定义如下：

```
class CMSComm : public CWnd
{
protected:
```

```
 DECLARE_DYNCREATE(CMSComm)                        //动态创建宏
public:
...                                                //省略部分定义
virtual BOOL Create(LPCTSTR lpszClassName,
       LPCTSTR lpszWindowName, DWORD dwStyle,
       const RECT& rect,
       CWnd* pParentWnd, UINT nID,
       CCreateContext* pContext = NULL)
{return CreateControl(GetClsid(), lpszWindowName, dwStyle, rect, pParentWnd,
nID); }
                                                   //创建串口控件对象
    BOOL Create(LPCTSTR lpszWindowName, DWORD dwStyle,
     const RECT& rect, CWnd* pParentWnd, UINT nID,
     CFile* pPersist = NULL, BOOL bStorage = FALSE,
     BSTR bstrLicKey = NULL)
{
return CreateControl(GetClsid(), lpszWindowName, dwStyle, rect, pParentWnd,
nID,
       pPersist, bStorage, bstrLicKey);
}
public:
 void SetCommID(long nNewValue);
 long GetCommID();
 void SetCommPort(short nNewValue);                //设置串口号码
 short GetCommPort();                              //获得串口号码
 void SetCTSHolding(BOOL bNewValue);               //设置 CTS 的状态
 BOOL GetCTSHolding();
 void SetDSRHolding(BOOL bNewValue);               //设置 DSR 的状态
 BOOL GetDSRHolding();
 void SetDTREnable(BOOL bNewValue);                //设置 DTR 的状态
 BOOL GetDTREnable();
 void SetHandshaking(long nNewValue);              //设置通信握手方式
 long GetHandshaking();                            //获取通信握手方式
 void SetInBufferSize(short nNewValue);            //设置输入缓冲区大小
 short GetInBufferSize();                          //获取输入缓冲区大小
 void SetInBufferCount(short nNewValue);
 short GetInBufferCount();
 void SetBreak(BOOL bNewValue);                    //设置是否包含停止位
 BOOL GetBreak();
 void SetInputLen(short nNewValue);                //设置读取接收数据的字节数
 short GetInputLen();                              //获取读取接收数据的字节数
 void SetNullDiscard(BOOL bNewValue);
 BOOL GetNullDiscard();
 void SetOutBufferSize(short nNewValue);           //设置输出缓冲区的长度
 short GetOutBufferSize();                         //获取输出缓冲区的长度
 void SetOutBufferCount(short nNewValue);
 short GetOutBufferCount();
 void SetParityReplace(LPCTSTR lpszNewValue);
 CString GetParityReplace();
 void SetPortOpen(BOOL bNewValue);                 //打开或关闭串口，设置为 TRUE 表示打开
 BOOL GetPortOpen();                               //串口是否已打开，如果为 TRUE 表示打开
 void SetRThreshold(short nNewValue);              //设置串口缓冲区中的字符数，以便产
                                                   生事件
 short GetRThreshold();
 void SetSettings(LPCTSTR lpszNewValue);           //设置串口，如设置为 9600,n,8,1
 CString GetSettings();
 void SetSThreshold(short nNewValue);              //若为 0，则表示发送数据的
```

```
short GetSThreshold();
void SetOutput(const VARIANT& newValue);
                                              //一个非常重要的函数，用于写串口
 VARIANT GetOutput();
void SetInput(const VARIANT& newValue);
                                              //一个非常重要的函数，用于读串口
 VARIANT GetInput();
 void SetCommEvent(short nNewValue);
 short GetCommEvent();                  //获得串口事件
 void SetEOFEnable(BOOL bNewValue);     //设置结束标志位
 BOOL GetEOFEnable();                   //获得结束标志位
 void SetInputMode(long nNewValue);     //设置接收模式
 long GetInputMode();                   //获取接收模式
...                                    //省略部分成员函数
};
```

在串口类的头文件中，列出了比较常用的 CMSComm 类成员函数。其中，用户需要特别注意一下。例如，串口类对象的创建、初始化串口以及串口参数设置等。

注意：串口控件类的成员函数说明请参考上面的该类定义代码。

2．使用CMSComm类

在上面的小节中，向用户介绍了 CMSComm 类的头文件。用户在该类的头文件中，可以看到主要的成员函数声明等。在本小节中，将在程序中，使用该类进行相关的串口操作，并向用户介绍这些操作的方法。

（1）用户要使用 CMSComm 类，必须在程序中包含该类的头文件。代码如下：

```
#include MSComm.h"                        //包含 CMSComm 类头文件
...                                        //省略部分代码
```

（2）在程序中创建该类的实例对象。代码如下：

```
...                                        //省略部分代码
CMSComm comm.;                             //定义 CMSComm 类对象
```

当用户成功定义 CMSComm 类对象以后，便可以使用该对象调用类中的成员函数进行相关的操作了。

13.1.3　MSComm 控件串行通信编程方法

在程序中，用户可以使用已经定义好的 CMSComm 类对象，对该类中的成员函数进行调用以实现串口功能。在本节中，主要向用户介绍该类中，常用的一些成员函数的原型以及使用方法。

1．设置串口参数

首先，用户需要使用串口类对象调用函数 SetCommPort()设置将打开的串口号。该函数原型如下：

```
void SetCommPort(short nNewValue); //设置串口号码
```

该函数的作用是指定或设置将打开的串口号码。参数 nNewValue 表示设置的串口号。例如，用户在程序中，将使用串口“COM1”进行串口通信，则设置串口号的代码如下：

```
...                                         //省略部分代码
comm. SetCommPort(1);                       //设置串口号为"COM1"
```

然后，用户需要设置通过串口接收数据的类型。实现该功能的函数是 SetInputMode()，其原型如下：

```
void SetInputMode(long nNewValue);          //设置接收数据的类型
```

该函数的作用是设置通过串口接收数据的类型。参数 nNewValue 的取值决定了接收数据的类型，其取值如表 13.1 所示。

表 13.1　串口接收数据类型的取值

取　值	含　义
0	表示接收数据的类型是文本类型
1	表示接收的数据类型为二进制类型

例如，用户需要通过串口接收二进制数据，则应该将参数指定为“1”。代码如下：

```
...                                         //省略部分代码
comm. SetInputMode(1);                      //设置接收数据的类型
```

注意： 当用户设置串口数据接收类型时，为了更完整地接收数据，应该使用二进制数据类型进行接收。

用户设置串口的相关参数，可以调用函数 SetSettings()来实现。该函数原型如下：

```
void SetSettings(LPCTSTR lpszNewValue);
```

参数 lpszNewValue 表示与该串口相关的参数，其顺序依次是波特率、奇偶校验、数据位数、停止位数。例如，用户使用该函数设置串口的相关参数，代码如下：

```
CString str="9600,n,8,1";                   //定义并初始化参数字符串
comm. SetSettings(str);                     //设置串口参数
```

在上面的程序中，用户将波特率设置为 9600（默认值），n 表示无校验位，数据位为 8，停止位为 1。其中，设置奇偶校验位的取值如表 13.2 所示。

表 13.2　设置奇偶校验位的取值

取　值	含　义
n	无校验位
e	偶校验位
o	奇校验位

当串口缓冲区中，接收到数据时，串口控件会产生串口事件。但是，用户可以通过调用函数 SetRThreshold()设置是否产生该事件。其原型如下：

```
void SetRThreshold(short nNewValue);
```

该函数的功能是由参数 nNewValue 的取值决定的。如果该参数取值为 0，则表示不产生串口事件。如果取值为 1，则表示每接收到一个字符就会产生串口事件。例如，用户调用该改函数设置是否产生串口事件，其代码如下：

```
...                                       //省略部分代码
comm. SetRThreshold(1);                   //设置是否产生串口事件
```

在程序中，用户设置串口为每接收到一个字符就产生串口事件。接下来，用户需要设置读取串口数据时，从串口缓冲区中所读取的字节数。实现设置读取数据字节数的函数是 SetInputLen()。函数原型如下：

```
void SetInputLen(short nNewValue);
```

参数 nNewValue 表示用户需要从接收数据中读取的字节数。如果该参数为 0，则表示用户希望将串口缓冲区中的数据全部读取。

2. 打开串口

如果用户设置完串口的相关参数以后，便可以调用函数 SetPortOpen()将串口打开。该函数原型如下：

```
void SetPortOpen(BOOL bNewValue);
```

该函数的作用是打开串口。参数 bNewValue 表示是否打开串口，若为 true，则表示打开串口。与该函数相对应的函数是 GetPortOpen()，其作用是判断当前串口是否处于打开状态。如果当前串口处于打开状态，该函数会返回 true。否则，将返回 false。

例如，用户完成串口的参数设置后，调用该函数打开端口，代码如下：

```
...                                   //省略部分代码
if(! m_Comm.GetPortOpen())            //判断串口是否已经打开
{
     m_Comm.SetPortOpen(TRUE);        //如果处于关闭状态，则将端口打开
    }
else                                  //如果串口处于打开状态，则提示用户串口已
                                        经打开
{
    MessageBox("串口已经打开");        //提示用户串口已经打开
    }
```

注意： 用户在进行串口编程时，应当养成良好的习惯。在打开一个串口前，必须使用函数 GetPortOpen()判断当前串口的状态是打开还是关闭的。然后，再调用函数 SetPortOpen()决定是否打开串口。

3. 发送串口数据

如果用户成功打开串口，那么便可以通过该串口向另一方发送数据了。在 MFC 中，用户可以调用函数 SetOutput()进行数据的发送。该函数的原型如下：

```
void SetOutput(const VARIANT& newValue);
```

该函数的作用是通过串口发送数据。参数 newValue 表示将要发送的数据，其类型必须强制转换为 COleVariant 类型。否则，数据发送将失败。例如，用户调用该函数进行数据发送，代码如下：

```
...                                          //省略部分代码
char array[100];                             //定义发送字符数组
comm.SetOutput(COleVariant(array));          //发送指定字符数组
```

注意： 当用户使用该函数进行数据发送时，必须将这些数据强制转换为 COleVariant 类型。

4．接收串口数据

在 MFC 中，当串口缓冲区中有数据到来时，串口控件会发送一个串口消息到指定的窗口，并由窗口类中的对应消息响应函数进行处理。因此，用户接受串口数据的操作应该在串口消息响应函数中进行。

（1）用户需要为串口控件添加串口消息的响应函数，如图 13.10 所示。

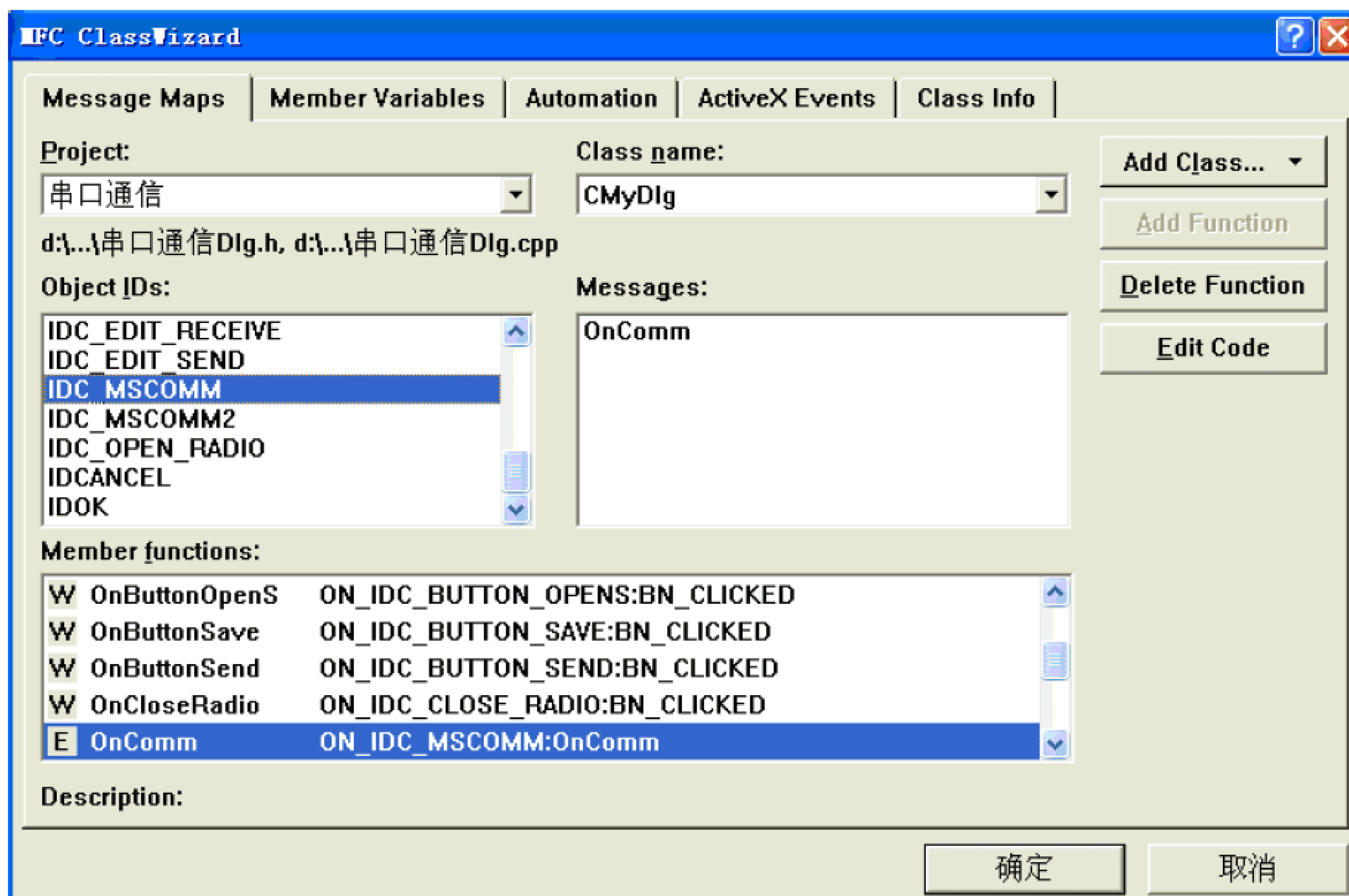


图 13.10　添加串口消息响应函数

（2）在该消息响应函数中实现接收串口数据的操作。代码如下：

```
 void CMyDlg::OnComm()                      //串口消息响应函数
{
 VARIANT variant;                           //定义 VARIANT 类型的变量
 COleSafeArray safearray;                   //定义 COleSafeArray 类对象
 int len;                                   //定义变量
 char rxdata[1000];                         //定义数组
 CString str;                               //定义字符串
 if(this->comm.GetCommEvent()==2)           //若发生的串口事件是读取事件
 {
           variant=this->comm.GetInput();   //读取串口缓冲区
```

```
            safearray=variant;                          //转换数据类型
            len=safearray.GetOneDimSize();              //获取有效数据的长度
            for(i=0;i<len;i++)
                  safearray.GetElement(&i,&rxdata[i]); //将数据转换为 Char
                                                          型数组
             str.Format("%c", rxdata);                  //格式化输出字符串
            }
                  MessageBox(str);                      //显示消息框
    }
```

当有串口事件发生时，用户首先判断该事件是否是串口读取事件。如果是，则使用串口类的函数 GetInput()读取缓冲区。再调用 COleSafeArray 类的函数 GetOneDimSize()获取有效数据的长度。最后，使用 COleSafeArray 类的函数 GetElement()读取数据并将该数据转换为 CHAR 类型。

注意： 当用户使用 MFC 串口控件进行串口通信编程时，必须遵循本节中所讲述的串口编程过程。

通过本节关于串口控件编程的学习，用户已经掌握了一般情况下，使用串口控件进行编程的基本步骤。所以，在 13.1.3 节中，将向用户介绍基于单文档应用程序的串口控件实例编程。

13.1.4　在基于单文档（SDI）程序中使用 MSComm 控件

在上面的小节中，已经向用户讲解了串口控件的添加、创建以及使用等方法。所以，在本节中，将利用 13.1.1 节中所创建的单文档实例工程，向用户讲解在基于单文档的应用程序中如何使用 MSComm（串口）控件。

1. 设计界面

首先，用户可以使用鼠标在面板中，放置控件并调整其位置，如图 13.11 所示。



图 13.11　实例程序界面设计

然后，实例程序界面中的各个控件 ID、属性以及作用如表 13.3 所示。

表 13.3　控件ID、属性以及作用

控　件　ID	属　　性	作　　用
IDC_COMNUM	下拉列表	显示可用的串口号
IDC_COMBO	下拉列表	显示波特率
IDC_COMJIAOYAN	下拉列表	选择校验方式
IDC_COMDATA	下拉列表	选择数据位
IDC_COMSTOP	下拉列表	数据保存方式
IDC_OPENCOM	按钮	打开串口
IDC_RECVDATA	按钮	接收数据
IDC_SAVADATA	按钮	保存数据
IDC_MSG	编辑框	显示接收到的数据
IDC_EDIT2	编辑框	编辑奖发送的数据
IDC_SENDDATA	按钮	发送数据

用户可以参考随书光盘中的实例界面，然后对比一下表 13.3 中所示的各个控件 ID 以及作用等。

2. 初始化界面

首先，该应用程序启动时，为了使程序窗口的大小固定。因此，用户需要在实例程序的框架类 CMainFrame 中实现禁用最大化按钮的功能。代码如下：

```
BOOL CMainFrame::PreCreateWindow(CREATESTRUCT& cs)
{
 if( !CFrameWnd::PreCreateWindow(cs) )      //调用基类的函数
        return FALSE;                       //返回 false
 cs.style&= ~(WS_MAXIMIZEBOX);              //将窗口样式的指定值相抵消掉
 return TRUE;                               //返回 true
}
```

在该功能的实现代码中，最为重要的一段代码是“cs.style&=～(WS_MAXIMIZEBOX);”，表示将禁用最大化按钮。运行以上代码，用户可以看到该程序窗口的最大化按钮已经被禁用了，如图 13.12 所示。

然后，用户还可以将窗口自带的状态栏和工具栏删除，并将程序窗口的图标设置为 QQ 图标。用户这样做，可以实现界面的美化。代码如下：

```
 int CMainFrame::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
 if (CFrameWnd::OnCreate(lpCreateStruct) == -1)     //调用基类的创建函数
      return -1;
 HICON icon=::AfxGetApp()->LoadIcon(IDI_ICON1);     //载入图标资源
 this->SetIcon(icon,true);                          //设置窗口图标
 return 0;
}
```

在上面的代码中，用户将 VC 编译器生成的关于状态栏和工具栏的相关代码删除。并调用函数载入图标资源以及设置程序窗口图标，如图 13.13 所示。

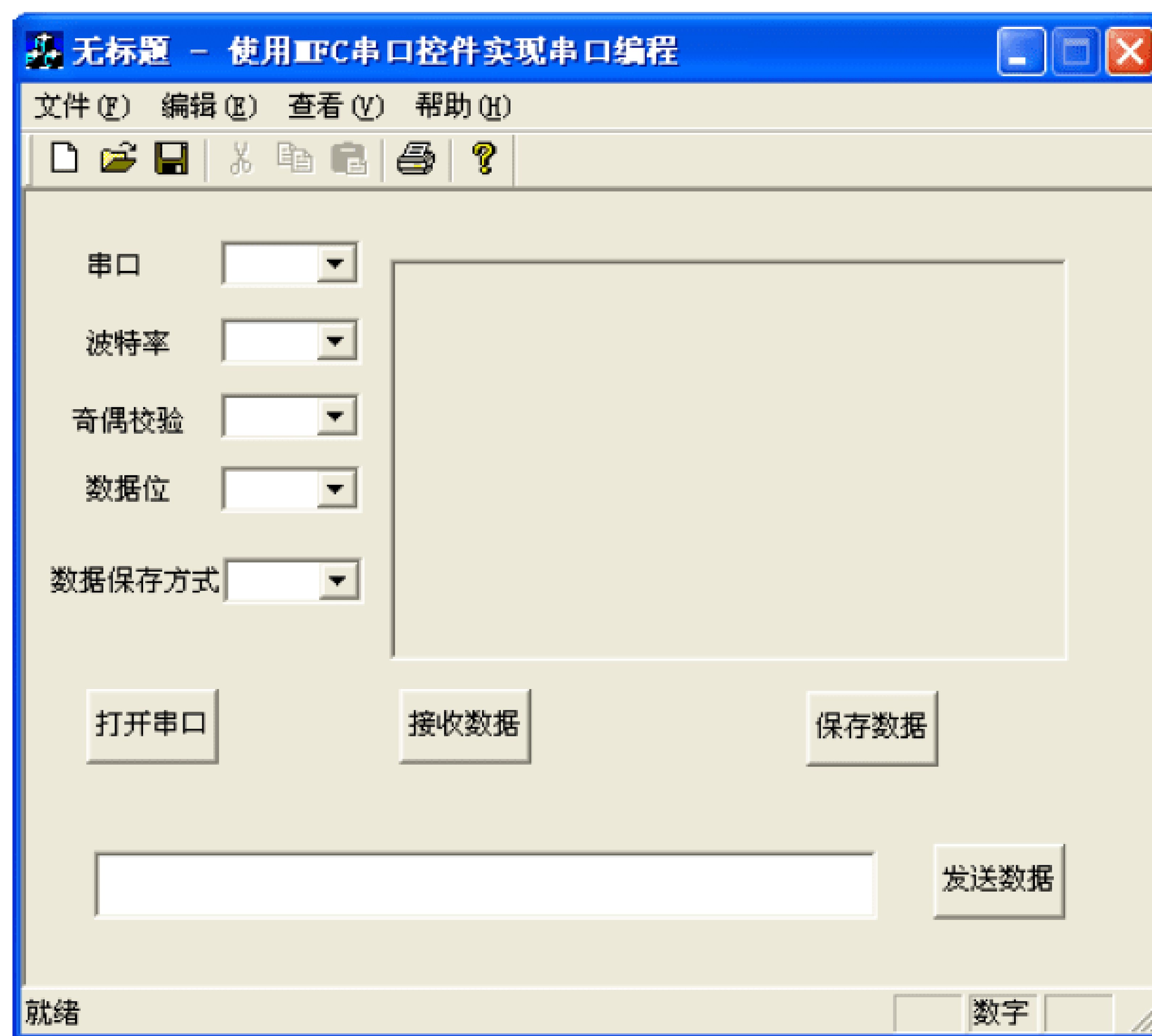


图 13.12　禁用窗口的最大化按钮

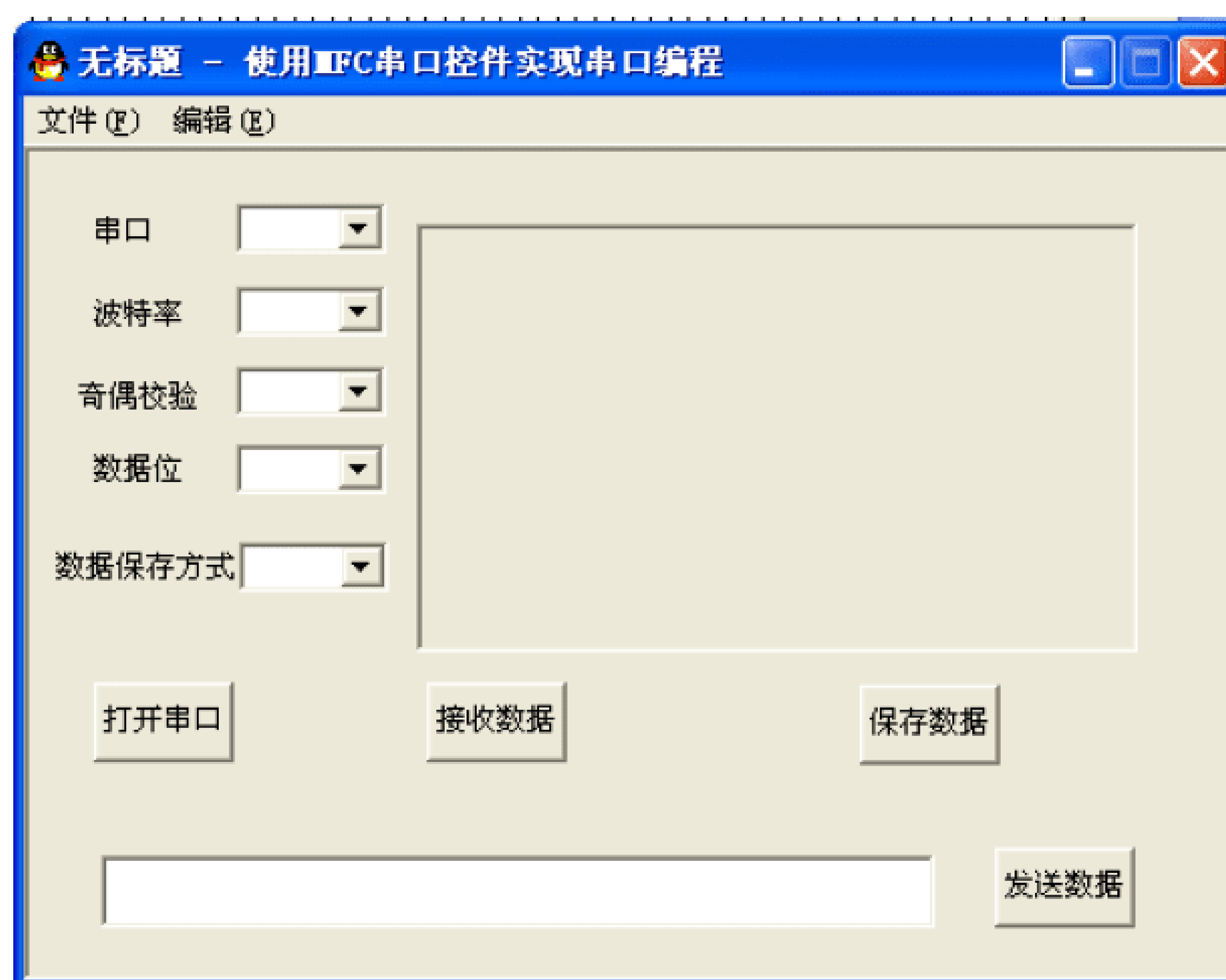


图 13.13　删除工具栏以及状态栏并设置窗口图标

注意：如果用户将函数 SetIcon 的第二个参数设置为 true，则表示将窗口的图标修改了。同时，还修改了该程序的大图标。用户可以打开该程序所在的目录，看到如图 13.14 所示的结果。

3．控件初始化

程序启动时，各个控件也应该同时初始化。例如，在下拉列表中，用户应该在控件初始化时，设置里面的选项，以便使用者进行选择。各控件的初始化工作可以在 CMFCView 类的初始化函数 OnInitialUpdate()中进行。

首先，用户使用快捷键 Ctrl+W，打开 MFC 应用程序向导为界面中的各个控件添加相应的变量，如图 13.15 所示。

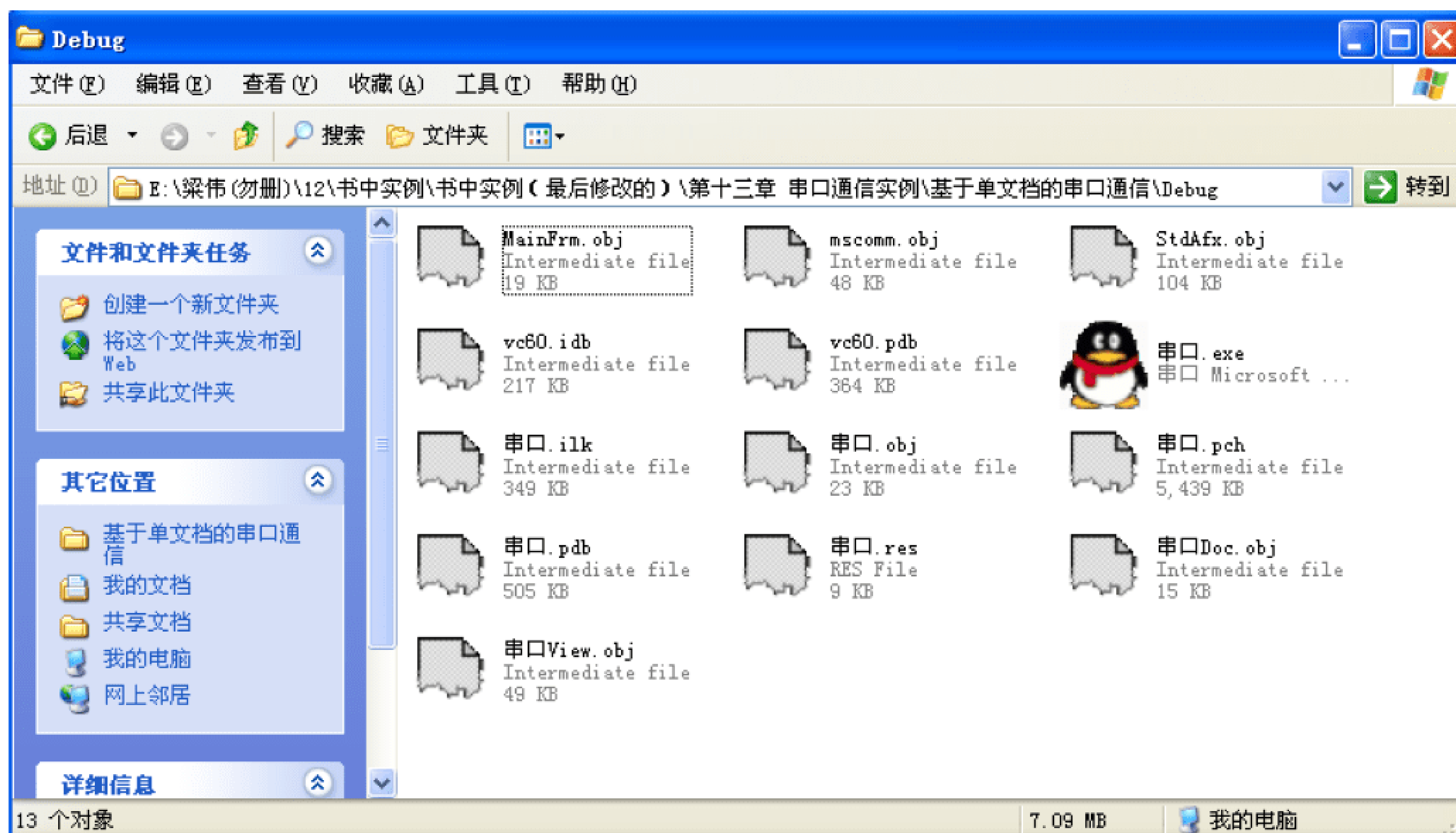


图 13.14　修改了程序的大图标显示

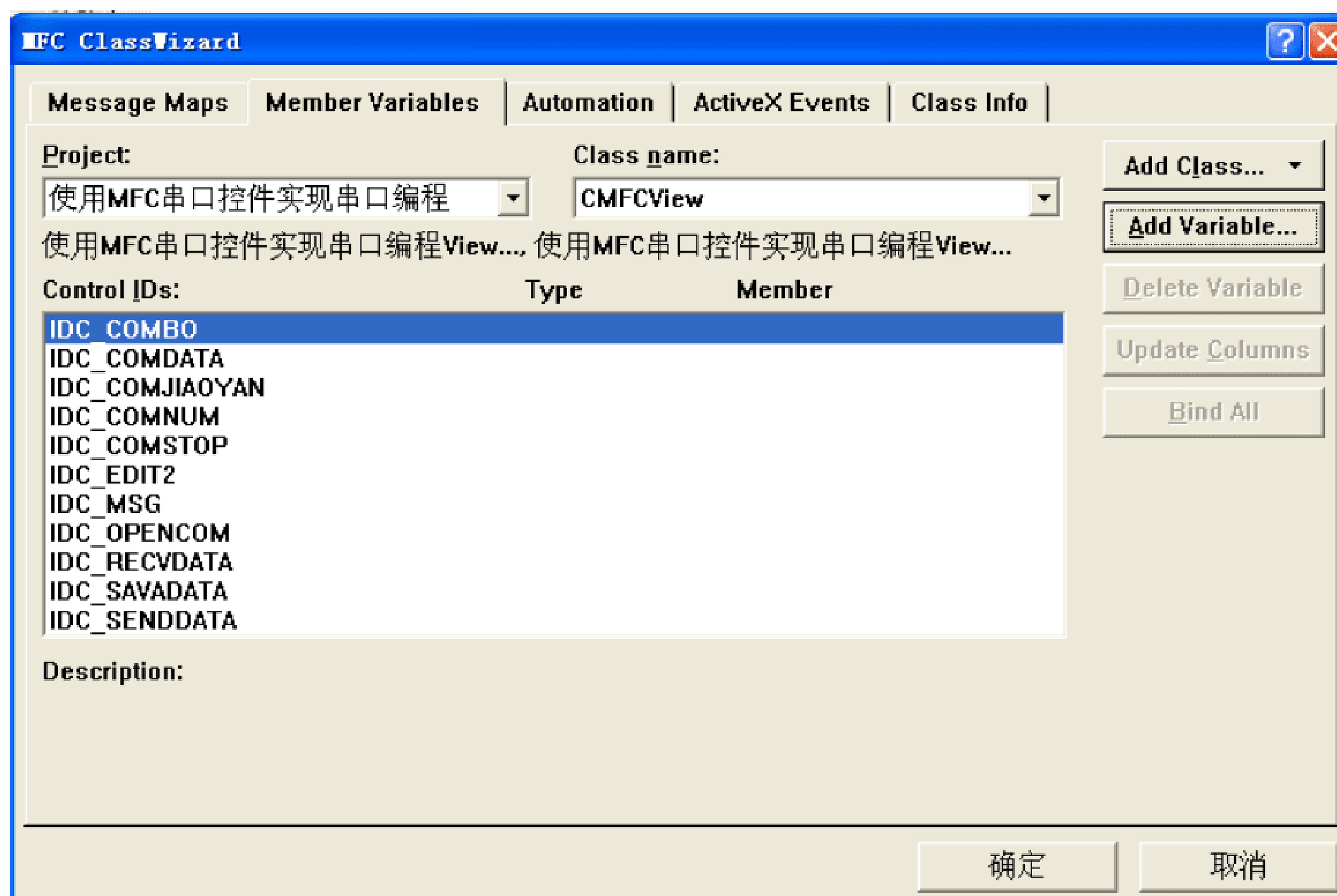


图 13.15　MFC 应用程序向导对话框

用户在 Control IDs 列表中选择将添加变量的控件 ID 后，单击 Add Variable 按钮弹出 Add Member Variable 对话框，如图 13.16 所示。

在图 13.16 所示的对话框中，用户需要为控件设置变量名以及类型。在该实例中，各个控件变量的类型必须指定为相应的控件类型。然后，按照这种方法为其他控件依次添加相应的控件变量。

然后，用户便可以在 CMFCView 类的初始化函数 OnInitialUpdate()中，使用这些控件变量调用相应的函数进行各自的初始化工作了。代码如下：

```
 void CMFCView::OnInitialUpdate()   //初始化函数
{
...                                 //省略部分代码
 m comnum.SetWindowText("COM1");    //初始化显示默认串口号
 m_comnum.AddString("COM1");        //向串口组合框中添加串口号字符串
 m comnum.AddString("COM2");
 m comnum.AddString("COM3");
 m comnum.AddString("COM4");
 m botelv.SetWindowText("1200");    //初始化显示默认波特率
 m_botelv.AddString("1200");        //向波特率组合框中添加波特率字符串
 m_botelv.AddString("2400");
 m botelv.AddString("9600");
 m_comjiaoyan.SetWindowText("奇校验");//初始化显示默认校验方式
 m_comjiaoyan.AddString("奇校验");  //向校验方式组合框中添加校验方式字符串
 m comjiaoyan.AddString("偶校验");
 m_comjiaoyan.AddString("无校验");
 m_comdata.SetWindowText("8");      //初始化显示默认数据位字符串
 m_comdata.AddString("4");          //向数据位组合框中添加数据位字符串
 m comdata.AddString("8");
 m_comstop.SetWindowText("文本");   //初始化显示默认保存数据方式字符串
 m comstop.AddString("文本");       //向数据保存方式组合框中添加保存方式
 m_comstop.AddString("不保存");
GetDlgItem(IDC_RECVDATA)->EnableWindow(false);
                                    //禁用各个按钮控件
GetDlgItem(IDC SAVEDATA)->EnableWindow(false);
GetDlgItem(IDC SENDDATA)->EnableWindow(false);
}
```

在上面的代码中，为了使控件初始化显示时显示默认值，所以，用户在各个控件添加字符串之前，均调用函数 SetWindowText()设置了默认值。将上面的代码保存、编译并运行，如图 13.17 所示。

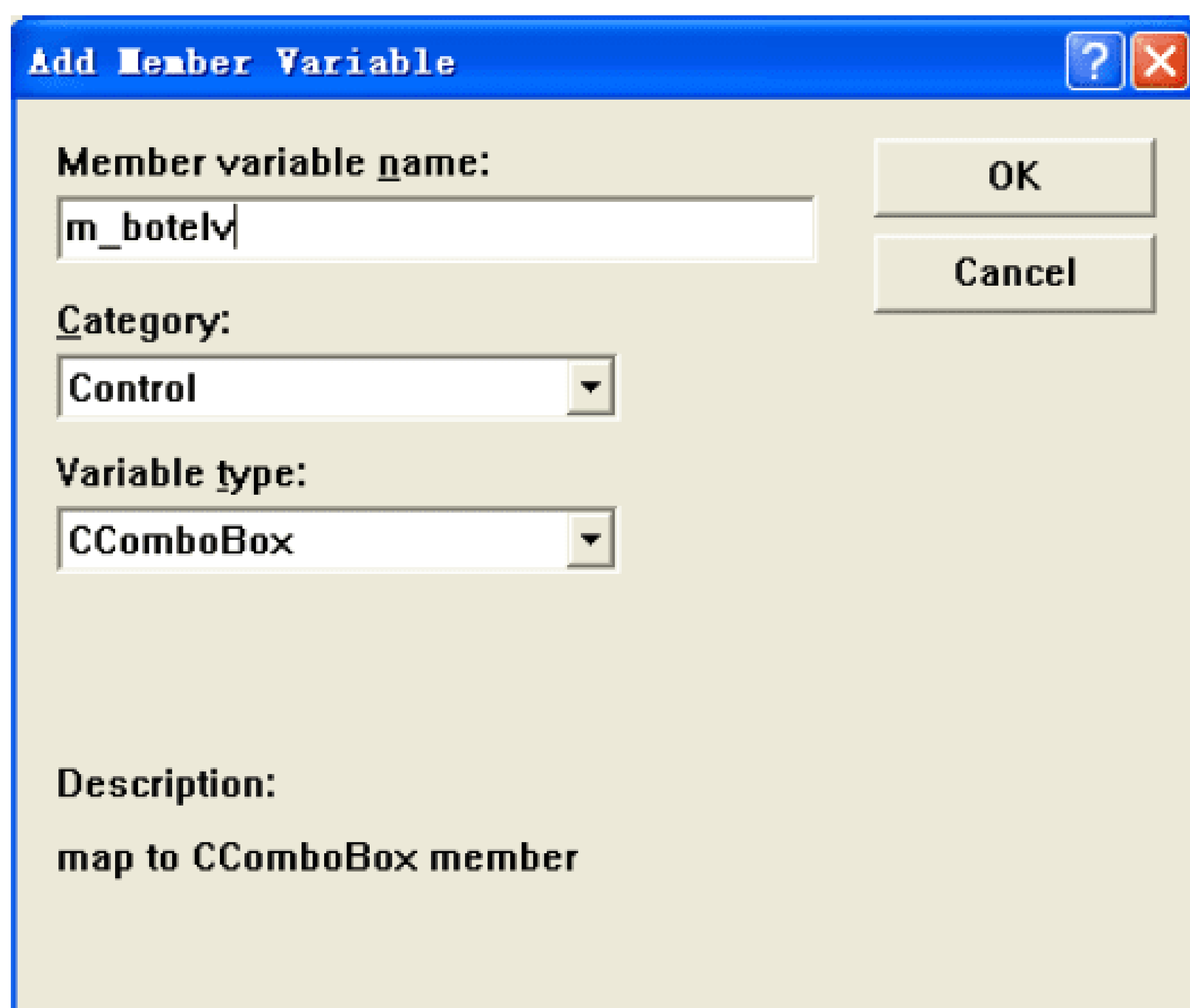


图 13.16　Add Member Variable 对话框

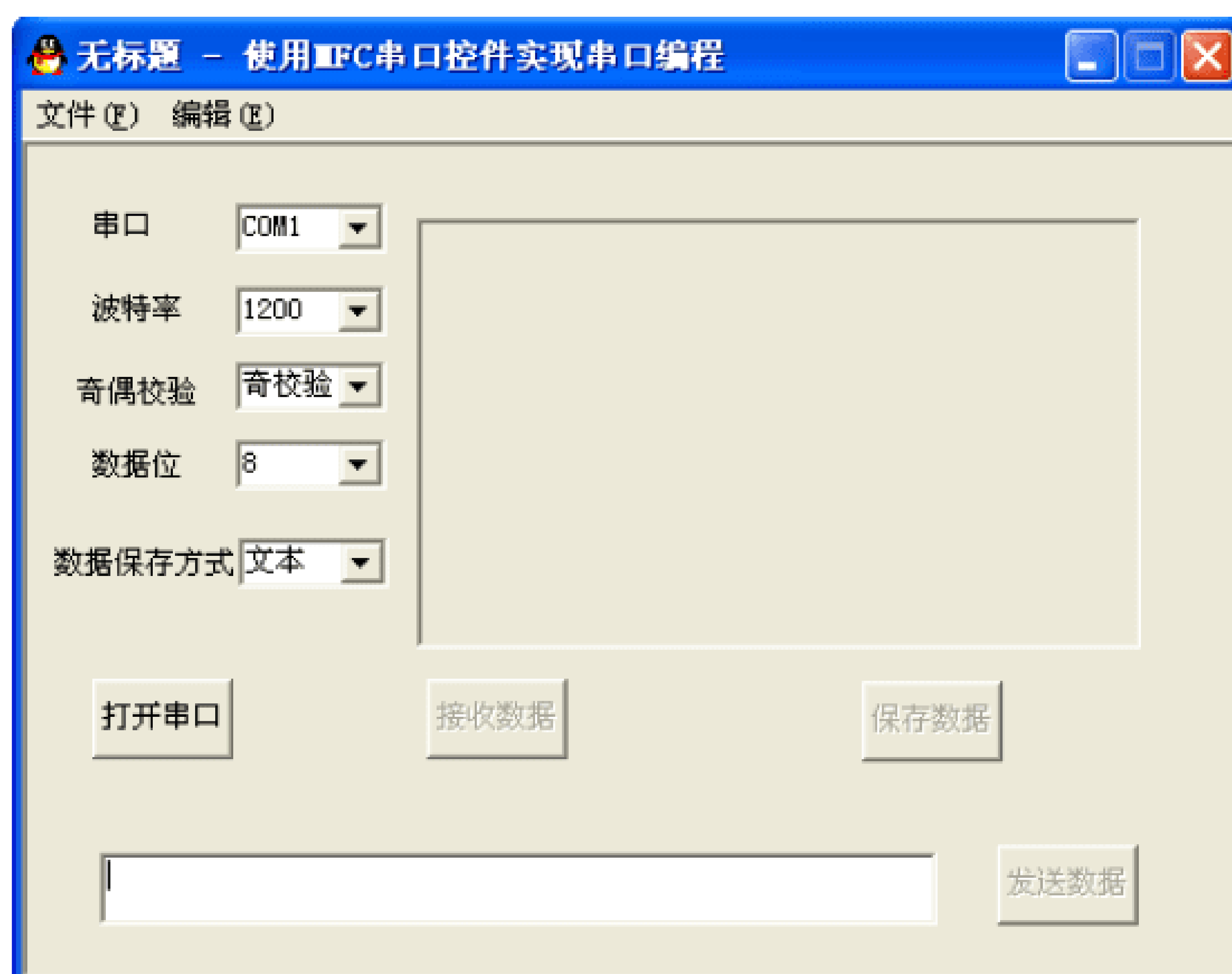


图 13.17　初始化控件后的界面显示

注意： 用户必须在打开串口成功之后，才可以使用其他功能。否则，其他功能根本毫无意义可言。

4．使用MSComm控件打开串口

如果用户将串口参数设置完毕以后，便可以单击“打开串口”按钮打开指定的串口了。首先，为该按钮添加消息响应函数 OnOpencom()。然后，用户在该函数中调用串口控件类的相关函数实现打开串口的功能。代码如下：

```
void CMFCView::OnOpencom()                  //打开串口按钮消息响应函数
{
     m_comnum.GetWindowText(str);           //获得串口号
 if(str.Find("COM1",0)!=-1)                 //判断串口号，以便关联相应的串口号
 {
       n=1;                                 //根据用户的选择设置相应的串口号
 }
 else
 {
     if(str.Find("COM2",0)!=-1)             //判断用户选择的串口号
     {
            n=2;
     }
     else
    {
         if(str.Find("COM3",0)!=-1)         //判断用户选择的串口号
         {
                n=3;
         }
        else
         {
                n=4;
         }
     }
}
str.Empty();                                //清空字符串
m_botelv.GetWindowText(str1);               //获取用户选择的波特率
str+=str1;                                  //连接字符串
str+=",";                                   //添加逗号
m_comjiaoyan.GetWindowText(str1);           //获取校验方式
if(str1.Find("奇校验",0)!=-1)               //判断用户所选择校验方式
      {
            str+="o";                       //字符 o 表示奇校验
      }
      else
      {
          if(str1.Find("偶校验",0)!=-1)     //判断用户所选择校验方式
          {
                 str+="e";                  //字符 e 表示偶校验
          }
          else
          {
             str+="n";                      //字符 n 表示无校验位
          }
      }

str+=",";
m_comdata.GetWindowText(str1);              //获取串口数据位
str+=str1;
str+=",";
```

```
str+="1";                               //设置串口数据停止位为1位
InitComm();                             //设置串口数据并打开指定串口
}
```

以上代码主要是用于获取各个控件中，用户所选择的串口参数。其中，函数 InitComm()是用户自定义的成员函数。其定义如下：

```
void CMFCView::InitComm()               //自定义函数 InitComm()
{                                       //代码中变量 mm 表示串口控件对象
 mm.SetCommPort(n);                     //将串口控件与指定的串口号相关联
 mm.SetInputMode(1);                    //设置接收数据的类型为文本类型
 mm.SetSettings(str);                   //设置串口的相关参数
 mm.SetRThreshold(1);                   //设置是否产生串口事件
 mm.SetInputLen(0);                     //是否全部读取串口缓冲区中的数据
    if(! mm.GetPortOpen())              //判断串口是否已经打开
{
    mm.SetPortOpen(true);               //如果处于关闭状态，则将端口打开
    }
}
```

用户将上面的程序保存、编译并运行，便可以看到程序运行的效果。如果用户计算机上的串口被占用或是不能使用，则程序会弹出错误提示，如图 13.18 所示。

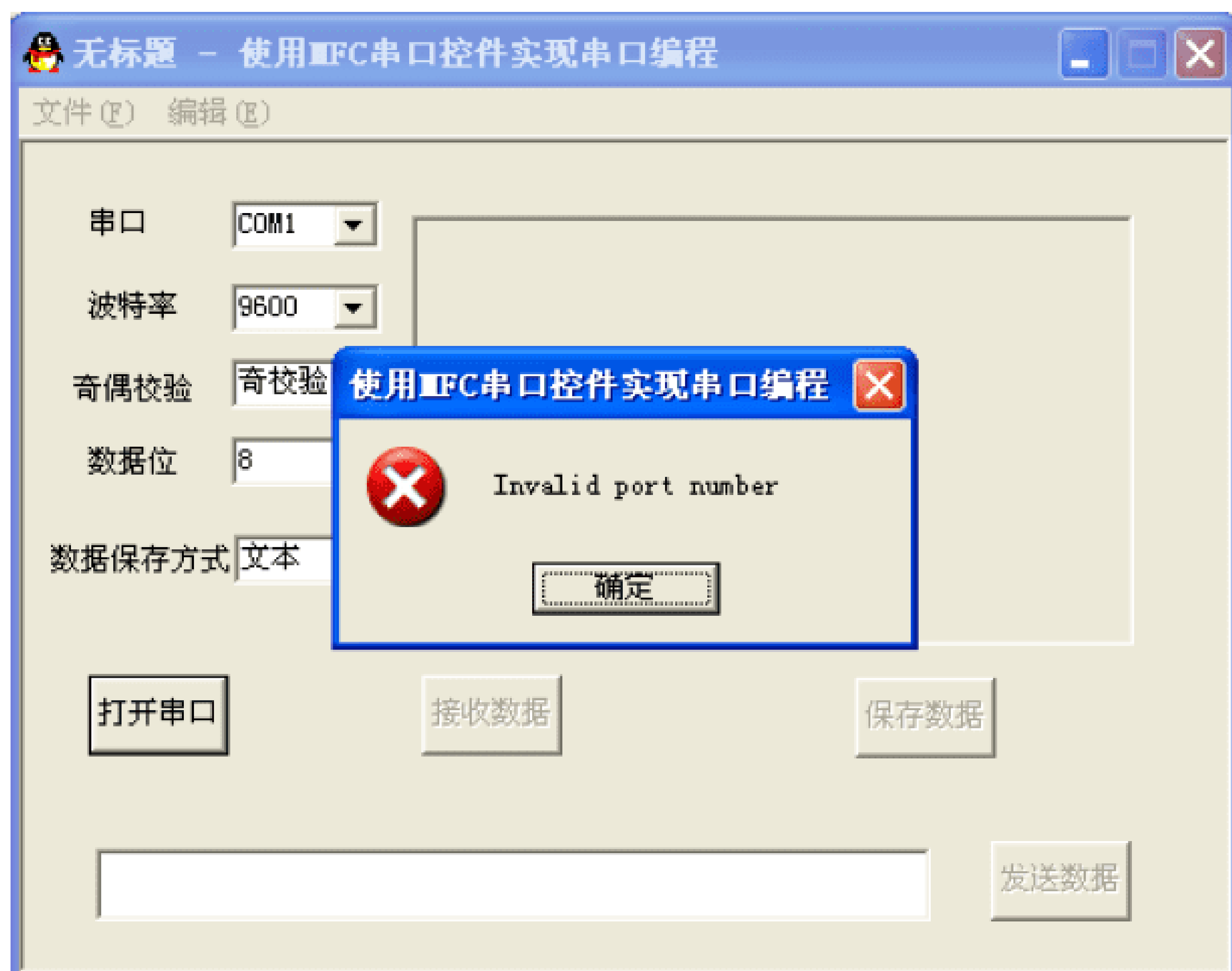


图 13.18　打开串口时发生错误

如果用户打开串口成功，则界面中的各个功能按钮均被激活，处于可用状态。代码如下：

```
...                                                  //省略部分代码
GetDlgItem(IDC_RECVDATA)->EnableWindow(true);        //设置各个功能按钮可用
GetDlgItem(IDC_SAVEDATA)->EnableWindow(true);
GetDlgItem(IDC_SENDDATA)->EnableWindow(true);
```

用户再次运行程序，成功打开串口后，界面中的按钮将处于可用状态，如图 13.19 所示。

在本节中，主要向用户讲解了 MSComm 控件在单文档应用程序中的参数设置以及初始化方法。通过本节的学习，用户应该可以掌握单文档界面的设计与初始化，以及 MSComm

控件的参数设置等。如用户未能完全理解，请复习第 12 章以及本节中的相关内容。

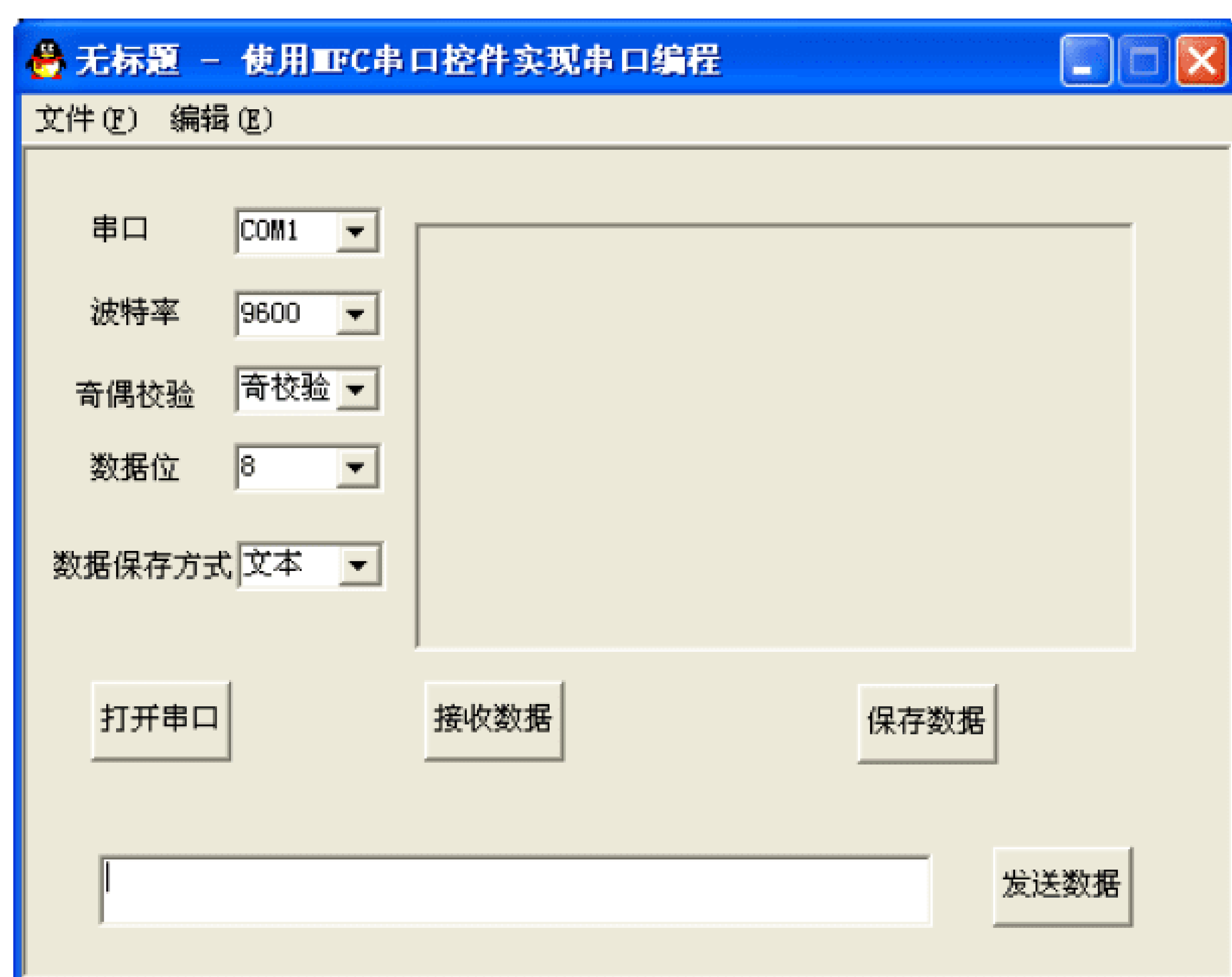


图 13.19　打开串口成功

13.1.5　应用 MSComm 控件控制串口实例

在上面的小节中，已经向用户讲解了串口参数的相关设置以及打开串口的方法。所以，在本节中，将向用户讲解如何通过串口发送数据与接收数据等编程方法。

1．发送串口数据

在 VC 平台下，如果用户使用串口控件进行串口通信编程。那么，用户需要在程序中调用串口控件类的成员函数 SetOutput()进行数据发送操作。但是，由于数据通过串口进行发送或接收时，必须将数据统一转换为 COleSafeArray 类型。否则，数据发送或接收将失败。

首先，用户使用 MFC 应用程序向导为“发送数据”按钮添加消息响应函数 OnSenddata()。然后，在该函数的定义中，使用串口控件对象 mm 调用其成员函数 SetOutput()将数据通过串口发送出去。代码如下：

```
void CMyView::OnSenddata()                         //发送数据按钮消息响应函数
{
 CString str;                                      //定义字符串变量
 char *a;                                          //定义字符指针
       GetDlgItem(IDC_EDIT2)->GetWindowText(str); //获取用户输入的数据
       if(str.GetLength()!=0)                      //判断用户输入是否为空
       {
              for(int i=0;i<str.GetLength();i++)  //根据输入数据的长度循环获取
                                                     各个字符
              {
                     a=str.GetBuffer(i);           //获取字符，并赋予字符指针
                     mm.SetOutput(COleVariant(a)); //调用串口数据发送函数发
                                                     送数据
                     memset(a,0,1);                //将字符指针赋 0 值
              }
```

```
        }
        else                                              //若用户输入的数据为空
        {
            MessageBox("输入数据不能为空！");              //提示用户不能输入空数据
        }
}
```

在上面的代码中，用户调用函数 GetWindowText()获取所输入的数据，并且判断该数据是否为空。如果数据为空，则提示用户该数据输入不能为空。否则，调用串口控件类的成员函数 SetOutput()将数据通过串口发送出去。当数据发送出去以后，用户需要将字符指针重新设置为 0。当用户输入数据为空或者没有输入数据时，单击“发送数据”按钮，程序会弹出错误提示，如图 13.20 所示。



图 13.20　用户未输入数据或输入数据为空则弹出错误提示

如果用户通过串口发送数据成功，则程序应该在接收消息编辑框中，显示数据发送成功的相关信息。所以，用户应该添加代码如下：

```
...                                          //省略部分代码
str1.Format("用户发送数据成功!\r\n");          //格式化输出字符串
GetDlgItem(IDC_MSG)->SetWindowText(str1);    //将该字符串输出到消息接收框中显示
```

那么，用户在发送数据成功以后，程序将在消息接收框中，显示数据发送成功信息，如图 13.21 所示。

在本节中，向用户讲解了使用串口控件通过串口发送数据的相关编程步骤以及方法。当然，用户在发送数据之前，最好再次判断一下，指定的串口是否真正被打开或是被占用。这样，会使程序使用更安全。

注意：用户在发送串口数据时，必须将数据类型强制转换为 COleVariant 类型。否则，数据将不能完整或安全地被发送或接收。

2．接收串口数据

用户使用串口控件进行串口通信编程，则当串口缓冲区中有数据到来时，该串口控件

会产生串口事件。在程序中，用户需要抓住串口控件的这一功能特性，在串口事件中，调用相关的函数对串口数据进行接收即可。

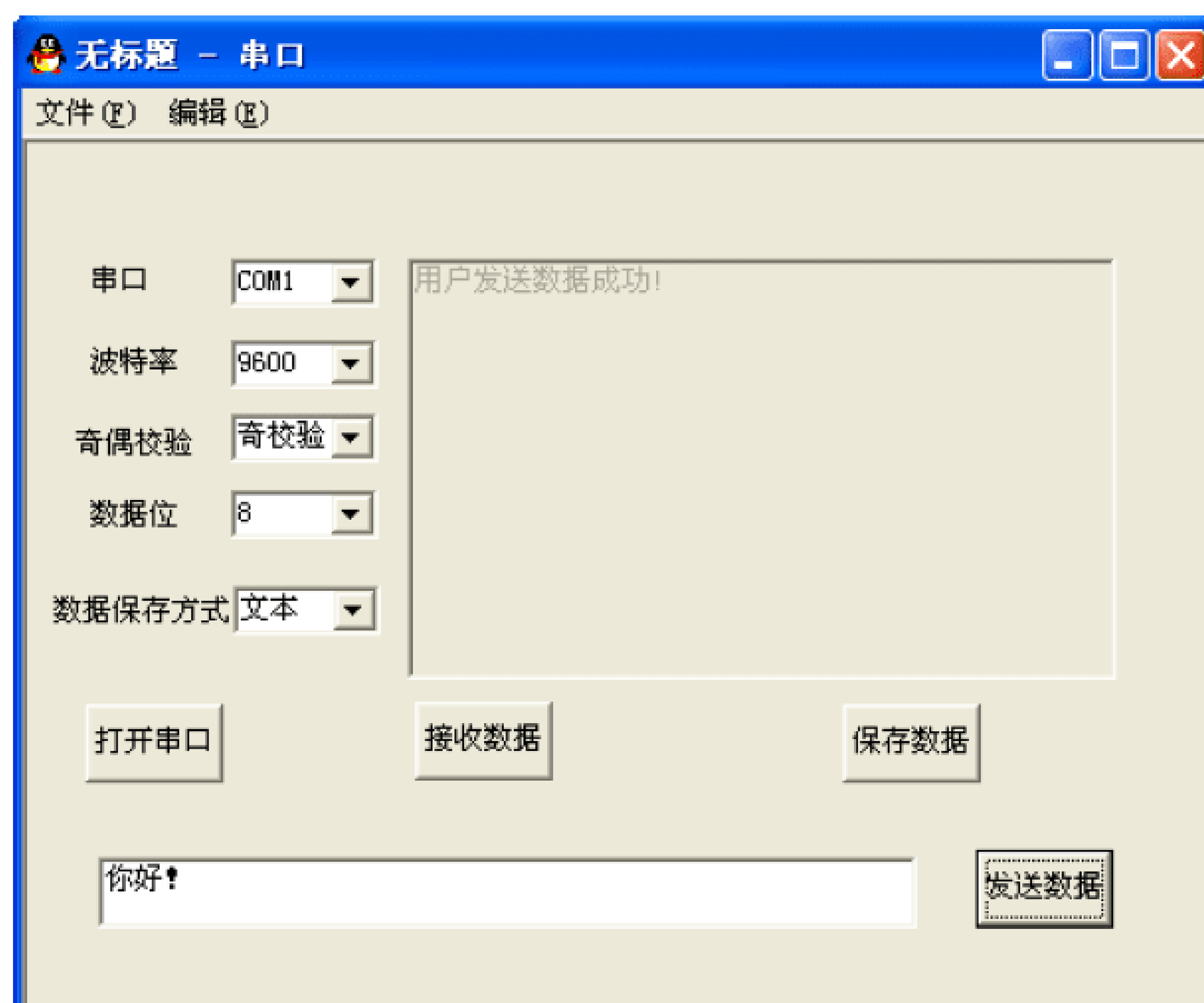


图 13.21　数据发送成功后的界面显示

首先，使用应用程序向导对话框为串口控件添加事件响应函数 OnComm()，如图 13.22 所示。

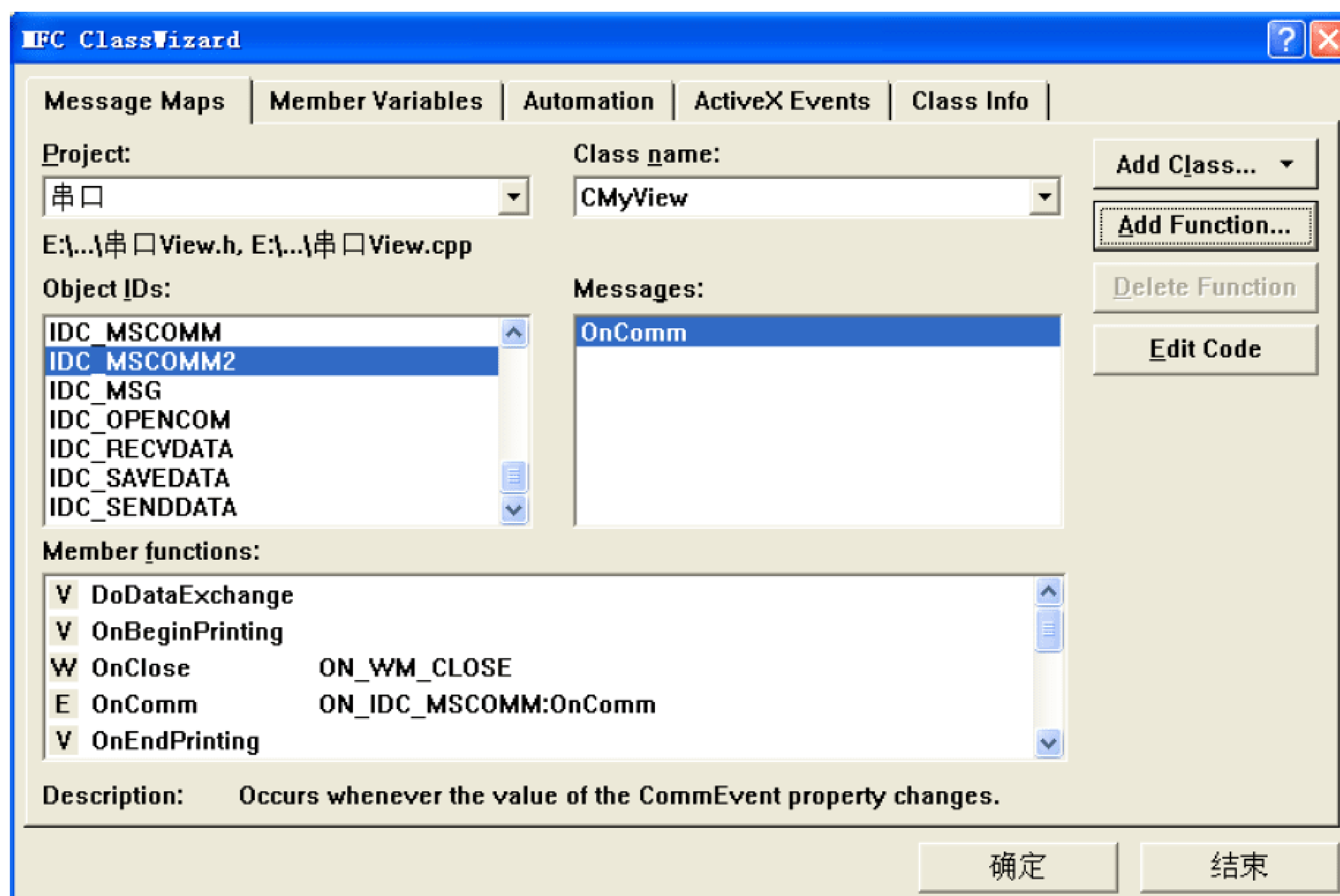


图 13.22　为串口控件添加事件响应函数

用户在控件 ID 列表中，选择串口控件 ID 以后，由于其仅有一个事件响应函数。因此，直接单击 Add Function 按钮弹出添加成员函数对话框。在该对话框中，用户可以修改串口事件响应函数的名称。在本书中，用户将该名称设置为 OnComm，然后单击 OK 按钮即可完成串口事件响应函数的添加。

实例程序中，用户接受串口数据的功能便可以在新添加的串口事件响应函数中实现。

其代码如下：

```
void CMyDlg::OnComm()                              //串口事件响应函数
{
 VARIANT variant;                                  //定义 VARIANT 类型的变量
 COleSafeArray safearray;                          //定义 COleSafeArray 类对象
 int len;                                          //定义变量
 char rxdata[1000];                                //定义数组
 CString str;                                      //定义字符串
 if(mm.GetCommEvent()==2)                          //若发生的串口事件是读取事件
 {
           variant= mm.GetInput();                 //读取串口缓冲区
           safearray=variant;                      //转换数据类型
           len=safearray.GetOneDimSize();          //获取有效数据的长度
           for(i=0;i<len;i++)
             safearray.GetElement(&i,&rxdata[i]);  //将数据转换为 CHAR 型数组
           str.Format("%c", rxdata);               //格式化输出字符串
            }
           MessageBox(str);                        //显示消息框
         }
```

在代码中，函数 GetCommEvent()的作用是获取当前发生的串口事件。如果该函数返回值为 2，则表示当前的串口事件为读取事件。

用户首先调用函数 GetCommEvent()判断当前发生的串口事件是否为读取事件。如果该函数返回值为 2，则说明当前发生的串口事件为读取事件。

接着，用户调用函数 GetInput()读取串口缓冲区中的数据，并返回该数据的类型。用户将返回的数据类型强制转换为 COleSafeArray 类型，并调用函数 GetOneDimSize()获取数据的有效长度。最后，使用函数 GetElement()将接收到的每个数据转换为 char 数组并将其格式化进行显示。

但是，在该实例中，需要将接收到的数据显示到消息接收框中。那么，用户只能将显示数据的工作交给界面中“接收数据”按钮的消息响应函数了。代码如下：

```
 void CMyView::OnRecvdata()
{
 CString str,str1;                                 //定义字符串
 GetDlgItem(IDC_MSG)->GetWindowText(str);          //获取消息接收框中原有的数据
 str1.Format("接收到的数据是：%c", rxdata);        //格式化接收到的数据
 str+="\r\n";                                      //添加换行符
 str+=str1;                                        //连接数据
 str+="\r\n";
 GetDlgItem(IDC_MSG)->SetWindowText(str);          //设置消息接收框中的数据
}
```

在代码中，变量 rxdata 是在前面所定义的字符数组变量，用于接收串口数据。如果用户按照这种方式显示数据，那么需要将该变量定义为工程视图类的成员变量。否则，用户使用该变量将发生错误。运行程序，单击“接收数据”按钮，程序会将接收到的数据显示在接收框中，如图 13.23 所示。

注意： 由于用户在通过串口发送数据时，是将数据类型强制转换为 COleSafeArray 类型的。所以，在接收数据时，需要使用 COleSafeArray 类的成员函数 GetElement()将接收到的数据转换为 char 型。否则，接收到的数据将不能被识别。

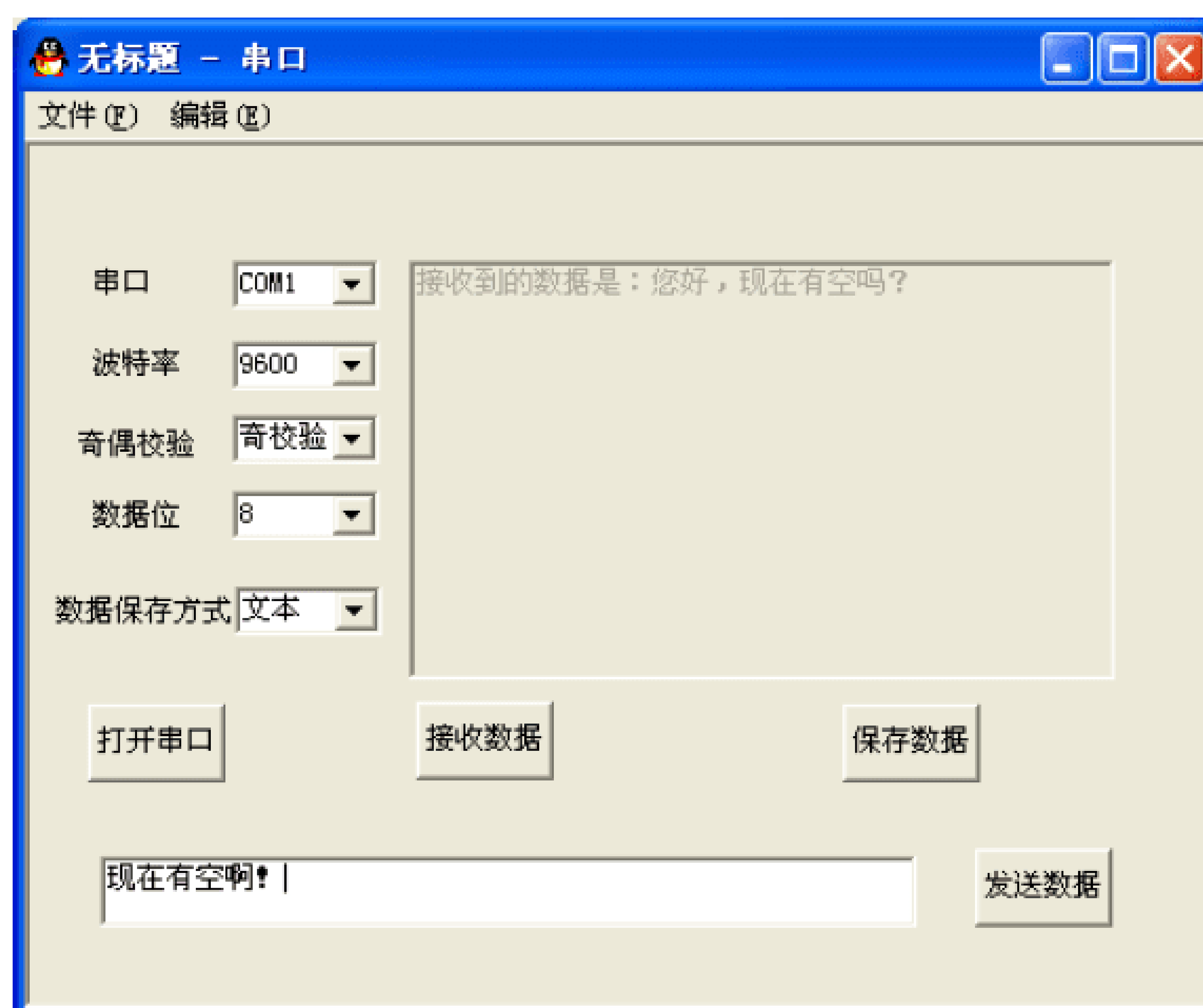


图 13.23　显示接收到的数据

在本节中，向用户讲解了在 VC 开发环境下，使用串口控件开发串口通信程序的步骤以及功能实现方法等。通过本节的学习，用户应该掌握了串口控件的常用属性以及功能函数的设置方法和使用方法。

13.2　串口 API 编程

在 13.1 节中，向用户讲解了在 VC 开发平台中，使用串口控件进行串口通信编程的基本步骤以及方法等。但是，在 Windows 平台下进行串口编程还可以使用 API（应用程序接口）函数实现。在本节中，将主要以串口 API 函数为主进行串口编程，并向用户讲解其原理以及方法。

13.2.1　Windows API 串口编程概述

在 Windows 平台下，串口其实可以被视为一种特殊的文件。那么，串口操作可以被视为一种文件操作。

用户在实际编程时，可以使用文件相关的 API 函数对串口进行关联或者操作。例如，CreateFile()、ReadFile()以及 WriteFile()等。如果用户使用函数 CreateFile()关联串口 COM1 并返回其句柄供后续操作使用。代码如下：

```
    HANDLE hModem;                              //定义串口句柄
...                                             //省略部分代码
hModem=CreateFile("COM1",GENERIC_READ|GENERIC_WRITE,0,0,OPEN_EXISTING,F
ILE_ATTRIBUTE_NORMAL,0);                        //关联串口并返回其句柄
    ...                                         //省略部分代码
```

在代码中，函数 CreateFile()将创建与串口 COM1 相关联的文件，并返回其句柄。用户对串口的后续操作，便可以使用该函数返回的文件句柄进行串口的相关操作。

注意：用户在创建与指定串口相关联的文件成功之后，还需要调用相关的 API 函数或者结构体为串口设置相应的参数值。

13.2.2　API 串口编程中用到的结构及相关概念说明

用户使用串口 API 函数进行串口通信编程时，需要用到一些相关的结构体或者是函数。在本节中，将向用户介绍这些结构体的定义、成员变量以及函数用法等。

1．串口编程相关结构体

与前面所讲的串口控件一样，使用串口 API 函数也需要为串口设置相关的参数。例如，波特率、校验方式等参数。但是，在串口 API 函数编程时，这些参数均被封装到了一个结构体中。该结构体定义如下：

```
typedef struct  DCB {
    DWORD DCBlength;                //该结构的大小
    DWORD BaudRate;                 //波特率
    DWORD fBinary: 1;               //是否选择停止位
    DWORD fParity: 1;               //是否产生串口事件
    DWORD fOutxCtsFlow:1;           //CTS 是否有效
    DWORD fOutxDsrFlow:1;           //DSR 是否有效
    DWORD fDtrControl:2;            //DTR 是否有效
    DWORD fDsrSensitivity:1;        //检测 DSR 是否接收到字符
    DWORD fTXContinueOnXoff:1;      //设置数据接收类型
    DWORD fOutX: 1;                 //设置是否在数据到来时启动流控制
    DWORD fInX: 1;                  //设置是否在发送数据时启动流控制
    DWORD fErrorChar: 1;            //是否使用奇偶校验替换错误
    DWORD fNull: 1;                 //是否将接收到的数据设置为空
    DWORD fRtsControl:2;            //RTS 流控制
    DWORD fAbortOnError:1;          //是否强行终止发生的错误
    DWORD fDummy2:17;               //该参数保留
    WORD wReserved;                 //不使用，必须设置为 0
    WORD XonLim;                    //根据该变量值确定发送数据最低的字节数
    WORD XoffLim;                   //指定发送数据的最小值
    BYTE ByteSize;                  //串口数据位大小
    BYTE Parity;                    //指定串口事件的类型
    BYTE StopBits;                  //设置停止位
    char XonChar;                   //接收和发送数据时，均需要设置该参数
char XoffChar;                      //接收和发送数据时，均需要设置该参数
char ErrorChar;                     //指定错误字符的替换值
    char EofChar;                   //数据结束字符
    char EvtChar;                   //设置事件字符
    WORD wReserved1;                //保留，以备使用
} DCB;
```

用户在实际编程时，需要填充该结构体以便设置串口的参数。在该结构体中，用户仅需要使用其中的几个即可。例如，串口的波特率、停止位等。其中，参数 BaudRate 的取值如表 13.4 所示。

表 13.4　参数BaudRate的常用取值

参 数 取 值	含　　义
CBR_110	表示将波特率设置为110
CBR_300	表示将波特率设置为300
CBR_1200	表示将波特率设置为1200
CBR_2400	表示将波特率设置为2400
CBR_4800	表示将波特率设置为4800
CBR_9600	表示将波特率设置为9600
CBR_14400	表示将波特率设置为14400
CBR_19200	表示将波特率设置为19200
CBR_38400	表示将波特率设置为38400

在表 13.4 中，已经向用户列举了部分常用的关于波特率的取值。在实际编程时，用户可以根据需要为串口选择合适的波特率。如果串口的波特率选择不合适，用户不但不能达到程序的预期效果，反而会使程序的稳定性受到影响。

参数 StopBits 表示停止位，其参数取值如表 13.5 所示。

表 13.5　参数StopBits的常用取值

参 数 取 值	含　　义
ONESTOPBIT	指定1个停止位
ONE5STOPBITS	指定1.5个停止位
TWOSTOPBITS	指定2个停止位

如果用户将该参数取值为 ONESTOPBIT，则表示用户将通信数据的停止位设置为 1 个字节。例如，用户将通信数据的停止位设置为 1 个字节。代码如下：

```
 DCB dcb;                          //定义 DCB 结构体变量
  dcb. DCBlength=sizeof(dcb);      //将该结构体的大小赋予成员变量
 dcb. StopBits= ONESTOPBIT;        //设置数据的停止位为 1 个字节
...                                //省略部分代码
```

用户在编写程序时，应该首先定义该结构体的变量。然后，再将该结构体的大小赋予变量 DCBlength，用户才能继续填充该结构体中的成员变量。例如，用户使用该结构体设置串口参数。代码如下：

```
...                                    //省略部分代码
 DCB dcb;
  dcb. DCBlength=sizeof(dcb);          //将该结构体的大小赋予成员变量
 dcb. BaudRate=9600;                   //指定串口数据传输的波特率
...                                    //省略部分代码
```

当用户将该结构体的各个常用变量填充完成之后，便可以调用相关的 API 函数进行串口参数设置了。

注意： 关于相关功能的 API 函数将在以下内容中向用户讲解。

在串口编程中，还有一个重要的结构体，即通信超时结构体。其具体定义如下：

```
typedef struct _COMMTIMEOUTS {
    DWORD ReadIntervalTimeout;                //数据读取超时设置
    DWORD ReadTotalTimeoutMultiplier;         //数据读取时间系数
    DWORD ReadTotalTimeoutConstant;           //数据读取时间常量
    DWORD WriteTotalTimeoutMultiplier;        //数据发送时间超时设置
    DWORD WriteTotalTimeoutConstant;          //数据发送时间常量
} COMMTIMEOUTS,*LPCOMMTIMEOUTS;
```

该结构体的主要作用是设置串口数据通信操作的超时设置。其中的成员变量都是以毫秒为单位。在通信过程中的总超时时间计算公式为：总超时=时间系数×要求读/写的字符数+时间常量。

例如，如果要读入 10 个字符，那么数据读取操作的总超时时间的计算公式为“ReadTotalTimeoutMultiplier * 10+ReadTotalTimeoutConstant”。从中可以看出，间隔超时和总超时的设置是不相关的，这可以方便串口通信程序灵活地设置各种超时时间间隔。

如果所有写数据超时的参数均设置为 0，表示不使用该超时时间间隔。如果 ReadIntervalTimeout 为 0，表示不使用数据读取超时。如果 ReadTotalTimeoutMultiplier 和 ReadTotalTimeoutConstant 都为 0，则表示不使用数据读取的总超时。如果数据读取间隔超时被设置成 MAXDWORD，并且将数据读取总超时设为 0。那么，用户在读取一次输入缓冲区中的数据后，不管是否读入了要求的字符，该读取操作都会被立即停止。

注意： 一般情况下，用户是不需要设置该结构体中的任何变量的，仅使用其默认超时间隔即可。但是，用户在一些对于时间要求非常严格的时候，便需要对该结构体进行详细的设置。

2. 串口编程的相关API函数

在本节中，将向用户重点介绍一些常用的串口 API 函数。并且将使用这些 API 函数进行程序示例的编写，使用户能够更深入地理解这些函数的用法。

首先，当用户使用函数 CreateFile()创建与指定串口相关联的文件成功后，便可以使用该函数返回的文件句柄进行串口参数设置。例如，用户为串口设置相关的参数。代码如下：

```
    HANDLE hModem;                                       //定义串口句柄
    hModem=CreateFile("COM1",GENERIC_READ|GENERIC_WRITE,0,0,OPEN_EXISTIN
G,FILE_FLAG_OVERLAPPED,0);                               //关联串口并返回其句柄
    DCB dcb;                                             //定义 DCB 结构体对象
    dcb. DCBlength=sizeof(dcb);                          //将该结构体的大小赋予成员变量
    dcb. BaudRate=9600;                                  //指定串口数据传输的波特率
        dcb.ByteSize =8;                                 //设置串口数据位大小为 8 个字节
    dcb.Parity = NOPARITY;                               //串口参数设置
    dcb.StopBits = ONESTOPBIT;
    dcb.fBinary = TRUE;
    dcb.fParity = FALSE;
```

然后，当用户将 DCB 结构体变量中的常用成员变量设置完成以后，便可以调用函数

SetCommState()为串口指定这些参数了。该函数的原型如下：

```
BOOL SetCommState(HANDLE hFile, LPDCB lpDCB );
```

该函数的作用是为串口指定相应的参数。其中，参数的含义如下：

- 参数 hFile 表示与串口相关联的文件句柄，也就是用户使用函数 CreateFile()时，所返回的句柄值。
- 参数 lpDCB 是指向结构体 DCB 的变量指针。

例如，用户使用该函数为串口设置相关的参数，其代码如下：

```
...                                          //省略部分代码
BOOL istrue;                                 //定义布尔变量
istrue=SetCommState(hModem, &dcb);           //调用函数进行参数设置
if(istrue)                                   //判断串口参数是否设置成功
{
        MessageBox("串口参数设置成功！");      //若参数设置成功，则提示用户
        }
else
{
    MessageBox("串口参数设置失败！请重试");   //若参数设置失败，则提示用户重试
    }
```

如果用户还需要为串口设置操作超时的时间间隔，那么实现该功能的 API 函数是 SetCommTimeouts()。该函数原型如下：

```
BOOL SetCommTimeouts(HANDLE hFile,LPCOMMTIMEOUTS lpCommTimeouts);
```

该函数的作用是为串口设置指定的操作超时间隔。其参数含义如下：

- 参数 hFile 表示与指定串口相关联的文件句柄。
- 参数 lpCommTimeouts 表示指向超时时间间隔结构体变量的指针。

例如，用户为串口设置操作超时时间间隔。代码如下：

```
...                                       //省略部分代码
COMMTIMEOUTS con;                         //定义结构体变量
con. ReadIntervalTimeout=10;              //设置串口数据读取的超时时间
BOOL istrue;                              //定义布尔变量
istrue= SetCommTimeouts(hModem, &con);    //调用函数进行参数设置
if(istrue)                                //判断串口参数是否设置成功
{
    MessageBox("超时时间设置成功！");       //若参数设置成功，则提示用户
    }
else
{
    MessageBox("超时时间设置失败！请重试"); //若参数设置失败，则提示用户重试
    }
...                                       //省略部分代码
```

在上面的代码中，用户将串口数据的读取超时时间设置为 10 毫秒。其表示当有数据到达串口缓冲区后，读取数据的线程开始从缓冲区中读取数据。如果在 10 毫秒内，该线程未能读取到任何数据，那么线程将返回。

如果用户没有为程序设置操作超时时间间隔，那么程序将可能发生假死现象。其实，这就是网络套接字中的异步模式。

接下来，用户需要为串口缓冲区指定大小。实现该功能的 API 函数是 SetupComm()。函数原型如下：

```
BOOL SetupComm(HANDLE hFile, DWORD dwInQueue,DWORD dwOutQueue);
```

该函数将为指定的串口缓冲区指定大小。其部分参数含义如下：

- ❑ 参数 dwInQueue 表示接收数据的缓冲区大小。
- ❑ 参数 dwOutQueue 表示发送数据的缓冲区大小。

例如，用户将串口的接收和发送数据缓冲区大小分别设置为 1024 和 512。代码如下：

```
...                                         //省略部分代码
SetupComm(hModem,1024,512);                 //设置各数据缓冲区的大小
...                                         //省略部分代码
```

在代码中，用户使用函数 SetupComm()将指定的串口数据缓冲区大小分别设置为 1024 和 512。

但是，当用户不使用串口缓冲区或者程序退出时，应该调用函数 PurgeComm()清除串口缓冲区中的所有内容。该函数原型如下：

```
BOOL PurgeComm(HANDLE hFile,DWORD dwFlags);
```

该函数执行成功将返回 true，否则，函数将返回 false。其参数含义如下：

- ❑ 参数 hFile 表示与串口相关联的文件句柄。
- ❑ 参数 dwFlags 表示串口缓冲区清除标志值。该标志值如表 13.6 所示。

表 13.6　串口缓冲区清除标志值

取　　值	含　　义
PURGE_TXABORT	禁止向接收缓冲区中写入数据
PURGE_RXABORT	禁止向发送缓冲区中写入数据
PURGE_TXCLEAR	清除接收缓冲区中的内容
PURGE_RXCLEAR	清除发送缓冲区中的内容

例如，用户将前面所指定的串口缓冲区中的内容清除。代码如下：

```
...                                         //省略部分代码
BOOL istrue;                                //定义布尔变量
istrue=PurgeComm(hModem, PURGE_TXABORT| PURGE_RXABORT| PURGE_TXCLEAR|
     PURGE_RXCLEAR);                        //调用函数对缓冲区内容进行清除
if(istrue)                                  //判断清除是否成功
{
        MessageBox("缓冲区数据清除成功！");    //若参数设置成功，则提示用户
        }
else
{
        MessageBox("缓冲区数据清除失败！请重试"); //若参数设置失败，则提示用户重试
        }
...                                         //省略部分代码
```

注意： 用户使用函数 PurgeComm()清除串口缓冲区中的内容时，必须同时为其指定标志 PURGE_TXABORT| PURGE_RXABORT，防止程序继续向缓冲区中读取或写入数据。

在本节中，主要向用户讲解了在串口通信编程中，常用的结构体以及函数的原型和使用方法等。同时，这些函数也是串口通信编程流程中的重要步骤。关于串口事件方面的知识，将在 13.2.3 节中向用户进行讲解。

13.2.3　串口通信事件

在串口通信中，如果指定的串口缓冲区有数据存在时，程序便会马上读取其中的数据。那么，程序是怎么知道缓冲区中有数据的？实际上，这个功能是由串口的硬件部分所实现的。也就是当数据到达缓冲区时，串口硬件会向操作该串口的应用程序发送相应的指令码。而应用程序则根据这个指令码进行相关的操作。本节将向用户讲解根据串口事件，对其进行读取和写入数据的实现方法。

1．串口事件

用户可以通过 API 函数为串口指定需要处理的串口事件。例如，是否接收到串口数据以及判断发送缓冲区中的最后一个字符是否为空等。实现这些功能的 API 函数是 SetCommMask()，其原型如下：

```
BOOL SetCommMask(HANDLE hFile, DWORD dwEvtMask);
```

该函数的作用是指定用户所感兴趣的串口事件。其参数含义如下：

- 参数 hFile 表示与串口相关联的文件句柄。
- 参数 dwEvtMask 表示相应的串口事件类型。其取值如表 13.7 所示。

表 13.7　串口事件类型

串口事件类型	含　义
EV_BREAK	收到BREAK信号所产生的串口事件
EV_CTS	CTS线路信号发生变化
EV_DSR	DSR线路信号发生变化
EV_ERR	线路状态错误，包括了CE_FRAME、CE_OVERRUN、CE_RXPARITY这3种错误
EV_RING	程序检测到响铃信号
EV_RLSD	CD线路信号发生变化
EV_RXCHAR	串口的输入缓冲区中已经接收到数据
EV_RXFLAG	使用SetCommState()函数设置的DCB结构中的等待字符已被传入输入缓冲区中
EV_TXEMPTY	串口的输出缓冲区中的数据已经全部被发送出去

在本书中，用户主要是通过检测缓冲区中，是否有数据到来或被输出，从而判断程序是否读取或发送缓冲区中的数据。所以，在表 13.7 中所示的串口事件类型，用户主要是使用串口事件 EV_RXCHAR 和 EV_TXEMPTY。例如，用户在程序中主要检测接收和发送事件，则使用函数 SetCommMask()进行功能实现。代码如下：

```
...                                                    //省略部分代码
BOOL istrue;                                           //定义布尔变量
istrue=SetCommMask(hModem, EV_RXCHAR | EV_TXEMPTY);
                                                       //设置串口事件的类型
if(istrue)                                             //判断串口事件是否成功
{
        MessageBox("串口事件设置成功！");               //若事件设置成功，则提示用户
     }
else
{
        MessageBox("串口事件设置失败！请重试");  //若事件设置失败，则提示用户重试
        }
...                                                    //省略部分代码
```

当用户设置串口事件的类型完成后，便可以等待串口事件的发生了。在 Windows 平台下，等待串口事件的发生是通过调用 API 函数 WaitCommEvent()实现的。该函数原型如下：

```
BOOL WaitCommEvent(HANDLE hFile,
          LPDWORD lpEvtMask,LPOVERLAPPED lpOverlapped);//指定等待的串口事件
```

该函数的作用是等待用户所指定的串口事件发生。如果用户指定的串口事件发生了，则该函数将返回 true。否则，函数将一直等待，直到有相应的串口事件发生为止。该函数有 3 个参数，其含义分别如下：

- 参数 hFile 表示与串口相关联的文件句柄。
- 参数 lpEvtMask 接收将要等待发生的串口事件。即当有串口事件发生时，该函数将把这个已经发生的事件写入该参数中。
- 参数 lpOverlapped 表示指向结构体 OVERLAPPED 的指针变量，用来保存程序异步操作的结果。

例如，用户调用该函数等待串口的接收事件发生。代码如下：

```
...                                                    //省略部分代码
BOOL istrue;                                           //定义布尔变量
DWORD de=0;                                            //定义事件保存变量
OVERLAPPED lpve;                                       //定义异步结构变量
istrue= WaitCommEvent (hModem,&de,&lpvc);              //等待事件的发生
if(istrue)                                             //判断事件等待是否成功
{
      ...                                              //省略部分代码
     }
```

在上面的代码中，用户首先调用函数 WaitCommEvent()等待串口事件的发生。如果有串口事件发生，则该函数返回 true。那么，用户便可以根据变量 de 的值，来判断发生的串口事件类型。所以，用户在上面代码的省略处添加代码如下：

```
if(istrue)                                             //判断事件等待是否成功
{
if ((dwEvtMask== EV_RXCHAR)                            //判断发生的串口事件是否为接收事件
      {                                                //如果是，则表示缓冲区中有数据到达
          ...                                          //用户需要在此处调用函数读取数据
         }
}
```

如果用户等待的串口事件与发生的串口事件相同，并且该事件是串口接收事件。那么，

用户可以调用函数 ReadFile()对串口缓冲区中的数据进行读取。关于串口数据读取相关的操作将在后面进行讲解。

本节主要向用户讲解了怎样实现设置串口事件类型以及等待串口事件的发生。并且当串口事件发生后，根据所发生的串口事件判断是否为用户所需要的串口事件。

注意： 本节中所讲解的串口事件的设置与相关处理，实际上与本书前面所讲的异步套接字事件的设置与处理是相同的，请用户回顾前面所讲的知识点。

2．接收串口数据

上面内容已经向用户讲解了怎样判断所发生的串口事件类型是串口的数据接收事件。下面将调用函数 ReadFile()对所发生的串口事件进行响应，并读取缓冲区中的数据。

首先，用户判断所发生的串口事件为数据接收事件后，需要调用函数 ClearCommError()清除串口错误并获取串口当前的状态，以便用户获取串口缓冲区中的数据大小。该函数原型如下：

```
BOOL ClearCommError(HANDLE hFile, LPDWORD lpErrors, LPCOMSTAT lpStat);
```

如果该函数调用成功，则返回 true。否则，该函数将返回 false。其参数及含义如下：

- ❑ 参数 hFile 表示与串口相关联的文件句柄。
- ❑ 参数 lpErrors 保存将要清除的串口错误。
- ❑ 参数 lpStat 是指向结构体 COMSTAT 的指针变量，用于保存串口当前的状态值。结构体 COMSTAT 的定义如下：

```
typedef struct  COMSTAT {
    DWORD fCtsHold:1;         //等待 CTS 信号的到来
    DWORD fDsrHold:1;         //等待 DSR 信号的到来
    DWORD fRlsdHold:1;        //等待 RLSD 信号的到来
    DWORD fXoffHold:1;        //等待 XOFF 字符的到来
    DWORD fXoffSent:1;        //等待 XOFF 字符的发送
    DWORD fEof:1;             //发送数据结束标志
    DWORD fTxim:1;            //等待字符
    DWORD fReserved:25;       //保留，不使用
    DWORD cbInQue;            //串口输入缓冲区中的字节数
    DWORD cbOutQue;           //串口输出缓冲区中的字节数
} COMSTAT, *LPCOMSTAT;
```

例如，用户获取串口输入缓冲区中的数据大小，则可以调用函数 ClearCommError()获取。如果获取成功，则输入缓冲区中的数据大小会被存放在结构体 COMSTAT 的成员变量 cbInQue 中。代码如下：

```
if ((dwEvtMask== EV_RXCHAR)                       //判断发生的串口事件是否为接收事件
      {                                           //如果是，则表示缓冲区中有数据到达
           DWORD doe;                             //定义清除的错误变量
            COMSTAT coms;                         //定义结构体变量
              ClearCommError(hModem,&doe,&coms); //清除串口事件并获取当前状态
      if(coms. cbInQue>0)                         //判断串口输入缓冲区中的数据大小
{
```

```
            ...                          //若大于 0，则表示接收到数据
          }
```

用户根据函数 ClearCommError()获取到的缓冲区数据大小，可以判断该缓冲区中是否有数据到来。

然后，用户可以调用函数 ReadFile()读取串口输入缓冲区中的数据。该函数原型如下：

```
BOOL ReadFile(
 HANDLE hFile,                          //与串口相关联的文件句柄
 LPVOID lpBuffer,                       //缓冲区，用于存放读取到的数据
 DWORD nNumberOfBytesToRead,            //需要读取的数据大小
 LPDWORD lpNumberOfBytesRead,           //实际读取到的数据大小
 LPOVERLAPPED lpOverlapped              //指向结构体的指针变量
);
```

由于该函数在前面的章节中已经向用户讲解了其用法与参数含义等。所以，在这里主要向用户讲解该函数的最后一个参数 lpOverlapped。在本节中，因为用户所创建的串口为异步通信模式。所以，用户必须将这里的参数设置为非空。代码如下：

```
if(coms. cbInQue>0)                       //判断串口输入缓冲区中的数据大小
{
     char buffer[coms. cbInQue]={0};      //定义并初始化缓冲区
      DWORD data;                         //存放实际读取到的字节数
       OVERLAPPED *over;                  //定义结构体指针变量
         ReadFile(hModem,&buffer, coms. cbInQue,data,over);
                                          //读取缓冲区中的数据
           }
```

在上面的代码中，用户调用函数 ReadFile()以异步方式对串口数据进行读取。所以，该函数不会等待其操作完成后才返回，而是立即返回 false。并且，用户使用函数 GetLastError()获取到的错误代码是 ERROR_IO_INCOMPLETE，表示该函数正在进行读取操作。

这时，用户可以调用函数 GetOverlappedResult()监视串口数据的大小变化。如果串口输入缓冲区中还有数据存在，则该函数将返回 false。否则，该函数将返回 true。那么，用户通过这一特性，便可以利用循环方式一直监视串口缓冲区中的数据变化情况，直到其数据为空。代码如下：

```
 BOOL f;                                              //定义布尔变量
 f=ReadFile(hModem,&buffer, coms. cbInQue,data,over); //读取缓冲区中的数据
 if(!f)                                               //判断读取函数的返回值
{
 BOOL m;                                             //定义布尔变量
 m=GetOverlappedResult(hModem, over, data,true);     //判断缓冲区中是否存在数据
 while(!m)                                          //如果存在，则进入循环
{
 if (GetLastError() == ERROR_IO_PENDING)            //判断错误代码
 {
      m=GetOverlappedResult(hModem, over, data,true); //继续调用函数监视缓
                                                        冲区数据
      }
```

```
            }
}
```

当程序循环完毕，则表示函数 ReadFile()已经将串口输入缓冲区中的数据全部读取。此时，用户便可以将缓冲区变量 buffer 中的数据显示出来即可。

3．发送串口数据

用户发送串口数据应该调用 API 函数 WriteFile()，由于本章所讲解的串口操作方式为异步通信方式。所以，用户在写入数据时，也需要将写操作设置为异步模式。因此，发送串口数据的基本步骤与接收串口数据的基本步骤相同。代码如下：

```
 BOOL f;                                           //定义布尔变量
 char buffer={"通过串口进行数据发送"};              //定义并初始化发送字符数组
 DWORD data=sizeof(buffer);                        //获取将写入数据的字节数
 DWORD data1,n=0;                                  //获取实际写入的字节数
 f=WriteFile(hModem,&buffer, data,data1,over);     //写入缓冲区中的数据
 if(!f)                                            //判断写入函数的返回值
{
 BOOL m;                                           //定义布尔变量
 m=GetOverlappedResult(hModem, over, data,true);  //判断缓冲区中是否剩余数据
 while(!m)                                         //如果有剩余，则进入循环
{
 if (GetLastError() == ERROR_IO_PENDING)          //判断错误代码
 {
      m=GetOverlappedResult(hModem, over, data,true); //继续调用函数监视缓
                                                          冲区数据
       n+=data1;                                   //将实际发送的数据进行累加
       if(n==data)                                 //实际发送的数据大小等于原大小
       {
            break;                                 //跳出循环
            }
       }
         }
}
```

4．使用多线程处理读操作

在程序中，为了避免出现程序等待的现象发生，用户可以使用多线程编程的方法将串口相关的写数据操作放在线程函数中进行。这样，不仅可以提高程序的运行效率，还可以使程序在非阻塞模式下运行。

首先，用户需要在程序中定义一个全局的线程函数并将其命名为 commdata。为了使该函数能在必要时向主窗口发送消息，所以该线程函数的参数设置为窗口句柄。该线程函数定义如下：

```
DWORD WINAPI commdata (HWND h);                    //线程函数定义
```

然后，在程序中调用函数 CreateThread()创建线程，并启动该线程。该函数原型如下：

```
HANDLE CreateThread(
 LPSECURITY_ATTRIBUTES lpThreadAttributes,         //指向结构体的指针变量
```

```
 DWORD dwStackSize,                              //初始化线程空间大小
 LPTHREAD_START_ROUTINE lpStartAddress,          //线程函数地址
 LPVOID lpParameter,                             //线程函数的参数
 DWORD dwCreationFlags,                          //创建标志
 LPDWORD lpThreadId                              //指定线程 ID
);
```

该函数创建线程成功，则返回该线程的线程句柄。否则，该函数将返回 false。如果该函数的参数 dwCreationFlags 设置为 0，则表示该线程创建成功立即运行。例如，用户创建串口操作线程。代码如下：

```
...                                               //省略部分代码
HANDLE hCommWatchThread;                          //定义线程句柄
CreateThread(NULL,0, & commdata,this->GetSafeHwnd(),0, &dwThreadID );
                                                  //创建线程
ASSERT(hCommWatchThread!=NULL);                   //判断线程句柄
...                                               //省略部分代码
```

在程序中，用户指定线程创建成功后，立即运行。接下来，用户需要实现线程函数的功能了。代码如下：

```
 DWORD WINAPI commdata (HWND h)                  //线程函数定义
{
...                                              //省略部分代码
 BOOL istrue;                                    //定义布尔变量
 DWORD de=0;                                     //定义事件保存变量
 OVERLAPPED lpve;                                //定义异步结构变量
 istrue= WaitCommEvent (hModem,&de,&lpvc); //等待事件的发生
 if(istrue)                                      //判断事件等待是否成功
 {
        if ((dwEvtMask== EV_RXCHAR)              //判断发生的串口事件是否为接收事件
         {                                       //如果是，则表示缓冲区中有数据到达
              DWORD doe;                         //定义清除的错误变量
               COMSTAT coms;                     //定义结构体变量
                ClearCommError(hModem,&doe,&coms);
                                                 //清除串口事件并获取当前状态
        if(coms. cbInQue>0)                      //判断串口输入缓冲区中的数据大小
        {
        char buffer[coms. cbInQue]={0};          //定义并初始化缓冲区
         DWORD data;                             //存放实际读取到的字节数
          OVERLAPPED *over;                      //定义结构体指针变量
             BOOL f;                             //定义布尔变量
 f=ReadFile(hModem,&buffer, coms. cbInQue,data,over); //读取缓冲区中的数据
 if(!f)                                          //判断读取函数的返回值
{
 BOOL m;                                         //定义布尔变量
 m=GetOverlappedResult(hModem, over, data,true);  //判断缓冲区中是否存在数据
 while(!m)                                       //如果存在，则进入循环
{
 if (GetLastError() == ERROR_IO_PENDING)         //判断错误代码
 {
       m=GetOverlappedResult(hModem, over, data,true); //继续调用函数监视缓
冲区数据
```

```
            }
                }
}
```

在程序中，函数 WaitCommEvent()在用户指定的串口上，等待相应的串口事件发生。若没有串口事件发生，则该函数将一直等待，直到有串口事件发生为止。若该函数所等待的串口事件发生，则该函数返回 true。

接下来，用户需要判断所发生的串口事件是否是串口接收事件。如果是串口接收事件，则调用函数 ClearCommError()获取串口缓冲区中的数据大小。并且根据数据大小定义缓冲区，使用函数 ReadFile()将这些数据全部保存到缓冲区中。由于在本章中，串口操作均为异步模式，所以函数 ReadFile()也采用异步模式。

注意： 在 while 循环中，用户可以使用函数 GetOverlappedResult()判断串口缓冲区中的数据是否被读取完毕。该函数原型如下：

```
 BOOL GetOverlappedResult(
   HANDLE hFile,                         //与串口相关联的文件对象句柄
   LPOVERLAPPED lpOverlapped,            //指向结构体 OVERLAPPED 的指针
   LPDWORD lpNumberOfBytesTransferred,   //指向实际读取或者写入串口的字节数
   BOOL bWait                            //标志
);
```

在该函数中，用户如果将第 4 个参数 bWait 设置为 true，则表示该函数将不会有任何的返回值，直到全部数据操作完成。如果将该参数设置为 false，则表示该函数在全部数据操作完成之前，每发生一次数据变化便返回 false。但是，用户可以使用函数 GetLastError()获取错误指令，如果该指令为 ERROR_IO_PENDING，则表示串口缓冲区中，还存在数据。当用户使用该函数时，一般是以循环方式调用该函数监视串口缓冲区等。

13.2.4　OVERLAPPED 异步 I/O 重叠结构

在前面的小节中，用户均使用异步模式创建与串口关联的文件，并对该文件进行读取和写入操作等。用户在编程时，使用异步模式对文件进行操作，可以大大提高程序运行的效率。异步编程与结构体 OVERLAPPED 有着密切的关系。所以，在本节中，将向用户介绍并讲解该结构体的定义以及用法。

当用户在创建文件或其他操作对象时，为其指定了相应的属性标志 FILE_FLAG_OVERLAPPED，则表示该操作对象是基于异步模式进行操作的。那么，在后续的对象操作中，都是基于异步模式进行。结构体 OVERLAPPED 的定义原型如下：

```
typedef struct _OVERLAPPED {               //结构体 OVERLAPPED 定义
    DWORD  Internal;
    DWORD  InternalHigh;
    DWORD  Offset;
    DWORD  OffsetHigh;
    HANDLE hEvent;
} OVERLAPPED;
```

结构体 OVERLAPPED 中，有 5 个参数，其作用以及含义如下。

- ❑ 参数 Internal：该参数由操作系统保留，表示与操作系统相关的状态。
- ❑ 参数 InternalHigh：表示发送或接收等所操作数据的数据长度。
- ❑ 参数 Offset：表示文件操作开始的位置。
- ❑ 参数 OffsetHigh：表示文件操作的字节偏移量。
- ❑ 参数 hEvent：表示文件操作后，将触发的事件句柄。

例如，用户在异步模式下，创建一个文件。代码如下：

```
    HANDLE hModem;                    //定义串口句柄
    hModem=CreateFile("COM1",GENERIC_READ|GENERIC_WRITE,0,0,OPEN_EXISTIN
G,FILE_FLAG_OVERLAPPED,0);     //创建异步模式文件并关联串口，返回其句柄
```

在代码中，用户创建与串口相关联的文件时，将其属性设置为 FILE_FLAG_OVERLAPPED，表示创建的该文件为异步访问模式。

异步访问模式的文件创建成功之后，用户使用函数 ReadFile()和 WriteFile()操作该异步模式文件时，都需要将这两个函数设置为异步模式。例如，用户使用这两个函数以异步模式对文件进行读写操作。代码如下：

```
...                                                         //省略部分代码
char buffer[coms. cbInQue]={0};                             //定义并初始化缓冲区
DWORD data;                                                 //存放实际读取到的字节数
OVERLAPPED *over;                                           //定义结构体指针变量
BOOL flag=ReadFile(hModem,&buffer, coms. cbInQue,data,over);
                                                            //读取缓冲区中的数据
if(!flag)                                                   //判断数据读取操作是否成功
{
        MessageBox("数据读取成功!");                        //提示用户数据读取成功
        }
else
{
        MessageBox("数据读取失败!");                        //若数据读取失败，则提示用户
        }
BOOL f=WriteFile(hModem,&buffer, data,data1,over); //写入缓冲区中的数据
if(!f)                                                      //判断数据写入是否成功
{
    MessageBox("数据写入成功!");                            //若数据写入成功，则提示用户
    }
else
{
     MessageBox("数据写入失败!");                           //若数据写入失败，则提示用户
     }
...                                                         //省略部分代码
```

在上面的代码中，用户使用函数 ReadFile()和 WriteFile()分别对文件进行读写操作。但是，值得用户注意的是在设置其参数时，其最后一个参数一定不能为 NULL。否则，这两个函数将不能以异步模式对文件进行读写。

在上面的例子程序中，用户还可以使用宏 READ_OS 和 WRITE_OS 对函数 ReadFile()和 WriteFile()的最后一个参数进行设置。这样，用户可以不再重新定义结构体 OVERLAPPED 的变量了。例如，使用宏 READ_OS 和 WRITE_OS 对文件操作设置异步模

式。代码如下：

```
...                                           //省略部分代码
istrue= WaitCommEvent (hModem,&de,&lpvc);    //等待事件的发生
if(istrue)                                    //判断事件等待是否成功
{
      if ((dwEvtMask== EV_RXCHAR)             //判断发生的串口事件是否为接收事件
      {                                       //如果是，则表示缓冲区中有数据到达
            DWORD doe;                        //定义清除的错误变量
             COMSTAT coms;                    //定义结构体变量
              ClearCommError(hModem,&doe,&coms); //清除串口事件并获取当前状态
      if(coms. cbInQue>0)                     //判断串口输入缓冲区中的数据大小
      {
      char buffer[coms. cbInQue]={0};         //定义并初始化缓冲区
       DWORD data;                            //存放实际读取到的字节数
        OVERLAPPED *over;                     //定义结构体指针变量
          BOOL f;                             //定义布尔变量
 f=ReadFile(hModem,&buffer, coms. cbInQue,data, READ_OS(npTTYInfo));
                                              //读取缓冲区中的数据
 if(!f)                                       //判断读取函数的返回值
{
 BOOL m;                                      //定义布尔变量
 m=GetOverlappedResult(hModem, READ_OS(npTTYInfo), data,true);
                                              //判断缓冲区中是否存在数据
 while(!m)                                    //如果存在，则进入循环
{
 if (GetLastError() == ERROR_IO_PENDING)      //判断错误代码
 {
      m=GetOverlappedResult(hModem, READ_OS(npTTYInfo), data,true);
                                              //继续调用函数监视缓冲区数据
       }
           }
 BOOL f=WriteFile(hModem,&buffer, data,data1, WRITE_OS(npTTYInfo));
                                              //写入缓冲区中的数据
 if(!f)                                       //判断数据写入是否成功
 {
    MessageBox("数据写入成功!");               //若数据写入成功，则提示用户
    }
else
{
    MessageBox("数据写入失败!");               //若数据写入失败，则提示用户
    }
...                                           //省略部分代码
```

注意： 如果用户使用宏 READ_OS 和 WRITE_OS 对函数 ReadFile()和 WriteFile()的最后一个参数进行设置，那么，在函数 GetOverlappedResult()中，同样需要将其中的相应参数设置为一样。否则，程序运行时，将发生逻辑错误。

在本节中，主要向用户讲解了结构体 OVERLAPPED 的定义以及各个成员变量的含义等。并且举例说明了在创建文件和读写文件时，使用该结构体的技巧与步骤等。用户可以通过本节的知识，编写代码打造自己的异步文件操作程序。

13.2.5　Win32 API 串口通信编程的一般流程

用户在前面的学习中，已经对串口编程的相关过程以及方法有了进一步的了解。所以，本节将具体地向用户介绍在 Win32 环境下，使用 API 函数进行串口通信编程的一般流程。

1．打开串口

当程序初始化时，用户需要打开串口并创建与该串口相关联的文件。代码如下：

```
    HANDLE hModem;                //定义串口句柄
    hModem=CreateFile("COM1",GENERIC_READ|GENERIC_WRITE,0,0,OPEN_
    EXISTING,FILE_FLAG_OVERLAPPED,0);        //关联串口并返回其句柄
```

注意：用户在使用函数 CreateFile()创建与串口相关联的文件时，必须将该文件的相关属性设置为 FILE_FLAG_OVERLAPPED。否则，用户所创建的文件将不能实现异步操作。

2．设置串口参数

用户可以调用函数 SetCommState()对串口进行相应参数的设置。但是，用户需要首先对串口参数结构体 DCB 进行初始化操作。代码如下：

```
DCB dcb;                          //定义 DCB 结构体对象
dcb. DCBlength=sizeof(dcb);       //将该结构体的大小赋予成员变量
dcb. BaudRate=9600;               //指定串口数据传输的波特率
     dcb.ByteSize =8;             //设置串口数据位大小为 8 个字节
dcb.Parity = NOPARITY;            //串口参数设置
dcb.StopBits = ONESTOPBIT;
dcb.fBinary = TRUE;
dcb.fParity = FALSE;
```

然后，用户便可以使用函数 SetCommState()对串口参数进行设置了。代码如下：

```
BOOL istrue;                                    //定义布尔变量
 istrue=SetCommState(hModem, &dcb);             //调用函数进行参数设置
 if(istrue)                                     //判断串口参数是否设置成功
 {
        MessageBox("串口参数设置成功！");        //若参数设置成功，则提示用户
        }
else
{
        MessageBox("串口参数设置失败！请重试");//若参数设置失败，则提示用户重试
        }
```

3．设置操作超时时间间隔

用户设置完串口的相关参数后，应该对串口操作的时间间隔进行设置。这样，当串口操作的时间间隔超出用户所设置的时间时，操作函数将被强制返回，避免程序假死。其代码如下：

```
...                                         //省略部分代码
COMMTIMEOUTS con;                           //定义结构体变量
con. ReadIntervalTimeout=10;                //设置串口数据读取的超时时间
BOOL istrue;                                //定义布尔变量
istrue= SetCommTimeouts(hModem, &con);      //调用函数进行参数设置
if(istrue)                                  //判断串口参数是否设置成功
{
     MessageBox("超时时间设置成功！");       //若参数设置成功，则提示用户成功
     }
else
{
    MessageBox("超时时间设置失败！请重试");  //若参数设置失败，则提示用户重试
    }
```

在程序中，用户主要是依靠结构体 COMMTIMEOUTS 中的成员变量 ReadIntervalTimeout 对串口操作的超时时间间隔进行设置的。

4．设置串口缓冲区

现在，用户可以调用函数对串口的数据缓冲区进行设置，实现其功能的 API 函数是 SetupComm()。代码如下：

```
...                                         //省略部分代码
SetupComm(hModem,1024,512);                 //设置各数据缓冲区的大小
```

当用户在程序退出或者其他原因，不再需要使用串口缓冲区时，应该将其中的内容进行清除操作并析构该缓冲区。否则，当下次再使用时，程序将发生错误。代码如下：

```
...                                         //省略部分代码
BOOL istrue;                                //定义布尔变量
istrue=PurgeComm(hModem, PURGE_TXABORT| PURGE_RXABORT| PURGE_TXCLEAR|
      PURGE RXCLEAR);                       //调用函数对缓冲区内容进行清除
if(istrue)                                  //判断清除是否成功
{
      MessageBox("缓冲区数据清除成功！");     //若参数设置成功，则提示用户成功
      }
else
{
      MessageBox("缓冲区数据清除失败！请重试"); //若参数设置失败，则提示用户重试
       }
```

5．设置串口事件

接下来，用户便可以根据实际应用设置串口的事件，并调用函数 WaitCommEvent()对串口进行监视。若有相应的串口事件发生，则返回。否则，程序将在串口上一直等待事件的发生。代码如下：

```
...                                         //省略部分代码
BOOL istrue;                                //定义布尔变量
istrue=SetCommMask(hModem, EV RXCHAR | EV TXEMPTY);
                                            //设置串口事件的类型
if(istrue)                                  //判断串口事件是否成功
{
        MessageBox("串口事件设置成功！");     //若事件设置成功，则提示用户成功
        }
```

```
else
{
        MessageBox("串口事件设置失败！请重试");//若事件设置失败，则提示用户重试
        }
BOOL istrue1;                                   //定义布尔变量
 DWORD de=0;                                    //定义事件保存变量
 OVERLAPPED lpve;                               //定义异步结构变量
 Istrue1= WaitCommEvent (hModem,&de,&lpvc);    //等待事件的发生
 if(istrue1)                                   //判断事件等待是否成功
{
        MessageBox("串口事件设置成功！");       //若事件等待成功，则提示用户成功
        }
else
{
        MessageBox("串口事件设置失败！请重试"); //若事件等待失败，则提示用户重试
        }
```

在代码中，用户首先为串口设置串口事件。然后，在相应的串口上等待串口事件发生并进行相应的处理。

6．读写串口

通过以上几个步骤，关于串口的相关参数设置以及串口事件指定等操作已经基本完成。那么，用户便可以对串口进行读写操作了。其代码如下：

```
...                                             //省略部分代码
    if ((dwEvtMask== EV_RXCHAR)                 //判断发生的串口事件是否为接收事件
    {                                           //如果是，则表示缓冲区中有数据到达
          DWORD doe;                            //定义清除错误的变量
           COMSTAT coms;                        //定义结构体变量
             ClearCommError(hModem,&doe,&coms); //清除串口事件并获取当前状态
    if(coms. cbInQue>0)                         //判断串口输入缓冲区中的数据大小
    {
    char buffer[coms. cbInQue]={0};             //定义并初始化缓冲区
     DWORD data;                                //存放实际读取到的字节数
      OVERLAPPED *over;                         //定义结构体指针变量
        BOOL f;                                 //定义布尔变量
 f=ReadFile(hModem,&buffer, coms. cbInQue,data,over); //读取缓冲区中的数据
 if(!f)                                         //判断读取函数的返回值
{
 BOOL m;                                        //定义布尔变量
 m=GetOverlappedResult(hModem, over, data,true);//判断缓冲区中是否存在数据
while(!m)                                       //如果存在，则进入循环
{
 if (GetLastError() == ERROR_IO_PENDING)        //判断错误代码
 {
     m=GetOverlappedResult(hModem, over, data,true);
                                                //继续调用函数监视缓冲区数据
      }
          }
BOOL f=WriteFile(hModem,&buffer, data,data1, WRITE OS(npTTYInfo));
                                                //写入缓冲区中的数据
if(!f)                                          //判断数据写入是否成功
{
    MessageBox("数据写入成功!");                 //若数据写入成功，则提示用户成功
```

```
    }
else
{
    MessageBox("数据写入失败!");                //若数据写入失败，则提示用户失败
    }
   }
  }
}
```

在本节中，主要向用户介绍了基于 API 函数进行串口通信编程的基本流程。并结合前面所讲解的程序实例代码向用户讲解每个步骤的编程方法等。

13.2.6　同步串口读写数据

在前面的小节中，向用户讲解的串口操作等均是基于异步模式的。所以，为了使读者将异步和同步模式区分开来，本节将向用户讲解同步模式下的串口编程。

1．基本概念

同步模式是指程序中的代码是以顺序执行的，即后面的程序必须等待前面的程序全部执行并返回后，才能继续执行。例如，用户使用函数 ReadFile()对文件进行读取操作，只要该函数正在进行数据读取并没有返回时，后面的程序只能等待其返回。

用户使用这种模式编写的程序具有很大的局限性，容易造成程序的假死，从而破坏程序界面的友好性。因此，一般情况下，不建议用户使用这种模式进行编程。

2．使用同步方式对串口进行操作

用户仅仅需要将前面的实例程序进行修改，便可以实现同步模式下的串口操作。代码如下：

```
HANDLE hModem;                              //定义串口句柄
hModem=CreateFile("COM1",GENERIC READ|GENERIC WRITE,0,0,OPEN EXISTING,
FILE_ATTRIBUTE_NORMAL,0);                   //关联串口并返回其句柄
DCB dcb;                                    //定义 DCB 结构体对象
dcb. DCBlength=sizeof(dcb);                 //将该结构体的大小赋予成员变量
dcb. BaudRate=9600;                         //指定串口数据传输的波特率
     dcb.ByteSize =8;                       //设置串口数据位大小为 8 个字节
dcb.Parity = NOPARITY;                      //串口参数设置
dcb.StopBits = ONESTOPBIT;
dcb.fBinary = TRUE;
dcb.fParity = FALSE;
BOOL istrue;                                //定义布尔变量
istrue=SetCommState(hModem, &dcb);          //调用函数进行参数设置
if(istrue)                                  //判断串口参数是否设置成功
{
MessageBox("串口参数设置成功！");            //若参数设置成功，则提示用户成功
    }
else
{
MessageBox("串口参数设置失败！请重试");      //若参数设置失败，则提示用户重试
    }
COMMTIMEOUTS con;                           //定义结构体变量
```

```
con. ReadIntervalTimeout=10;                    //设置串口数据读取的超时时间
BOOL istrue;                                    //定义布尔变量
istrue= SetCommTimeouts(hModem, &con);          //调用函数进行参数设置
if(istrue)                                      //判断串口参数是否设置成功
{
MessageBox("超时时间设置成功！");                //若参数设置成功，则提示用户成功
     }
else
{
MessageBox("超时时间设置失败！请重试");          //若参数设置失败，则提示用户重试
    }
SetupComm(hModem,1024,512);                     //设置各数据缓冲区的大小
BOOL istrue;                                    //定义布尔变量
istrue=PurgeComm(hModem,PURGE_TXABORT|PURGE_RXABORT|PURGE_TXCLEAR|
PURGE_RXCLEAR);                                 //调用函数对缓冲区内容进行清除
if(istrue)                                      //判断清除是否成功
{
    `   MessageBox("缓冲区数据清除成功！");       //若参数设置成功，则提示用户成功
         }
else
{
    MessageBox("缓冲区数据清除失败！请重试");
                                                //若参数设置失败，则提示用户重试
    }
BOOL istrue;                                    //定义布尔变量
istrue=SetCommMask(hModem, EV_RXCHAR | EV_TXEMPTY);
                                                //设置串口事件的类型
if(istrue)                                      //判断串口事件是否成功
{
    MessageBox("串口事件设置成功！");             //若事件设置成功，则提示用户成功
     }
else
{
    MessageBox("串口事件设置失败！请重试");
                                                //若事件设置失败，则提示用户重试
    }
BOOL istrue1;                                   //定义布尔变量
DWORD de=0;                                     //定义事件保存变量
istrue1= WaitCommEvent (hModem,&de,NULL);
                                    //等待事件的发生必须将第三个参数设置为NULL
if(istrue1)                                     //判断事件等待是否成功
{
    MessageBox("串口事件等待成功！");             //若事件等待成功，则提示用户成功
DWORD de=0;                                     //定义事件保存变量
if(istrue)                                      //判断事件等待是否成功
{
    if ((dwEvtMask== EV_RXCHAR)                 //判断发生的串口事件是否为接收事件
    {                                           //如果是，则表示缓冲区中有数据到达
          DWORD doe;                            //定义清除的错误变量
           COMSTAT coms;                        //定义结构体变量
            ClearCommError(hModem,&doe,&coms);
                                                //清除串口事件并获取当前状态
    if(coms. cbInQue>0)                         //判断串口输入缓冲区中的数据大小
    {
    char buffer[coms. cbInQue]={0};             //定义并初始化缓冲区
     DWORD data;                                //存放实际读取到的字节数
```

```
      BOOL f;                                    //定义布尔变量
 f=ReadFile(hModem,&buffer, coms. cbInQue,data,NULL);
                                                 //读取缓冲区中的数据
 if(!f)                                          //判断读取函数的返回值
{
 BOOL m;                                         //定义布尔变量
 m=GetOverlappedResult(hModem, NULL, data,true);
                                                 //判断缓冲区中是否存在数据
 while(!m)                                       //如果存在，则进入循环
{
 if (GetLastError() == ERROR_IO_PENDING)         //判断错误代码
 {
   m=GetOverlappedResult(hModem, NULL, data,true);
                                                 //继续调用函数监视缓冲区数据
        }
            }
        }
      }
}
     }
...                                              //省略部分代码
BOOL f=WriteFile(hModem,&buffer, data,data1, NULL);
                                                 //写入缓冲区中的数据
if(!f)                                           //判断数据写入是否成功
{
    MessageBox("数据写入成功!");                 //若数据写入成功，则提示用户成功
    }
else
{
    MessageBox("数据写入失败!");                 //若数据写入失败，则提示用户失败
     }
else
{
    MessageBox("串口事件等待失败! 请重试");
                                                 //若事件等待失败，则提示用户重试
    }
```

在上面的代码中，用户创建文件时，必须将该函数所创建的文件属性设置为 NULL。然后，当使用函数 ReadFile()和 WriteFile()对文件进行操作时，都需要将相应的参数设置为 NULL。否则，程序将不能对一个同步文件进行读写操作。

13.2.7　Win32 API 串口编程实例

本节将结合前面所讲的知识，向用户讲解如何使用 API 函数进行串口编程实例的相关方法。为了使用户能够区分 MFC 串口控件编程和 API 串口编程的不同之处，这里所使用的实例界面仍然是 MFC 串口控件编程所使用过的界面。

1. 打开串口

用户可以在界面中打开串口按钮的消息响应函数 OnOpencom()中，使用 API 函数将指定的串口打开。代码如下：

```
void CMyView::OnOpencom()                   //打开串口按钮消息响应函数
```

```
{
    HANDLE hModem;                          //定义串口句柄
hModem=CreateFile("COM1",GENERIC_READ|GENERIC_WRITE,0,0,OPEN_EXISTING,
FILE_ATTRIBUTE_NORMAL,0);                   //关联串口并返回其句柄
DCB dcb;                                    //定义 DCB 结构体对象
dcb. DCBlength=sizeof(dcb);                 //将该结构体的大小赋予成员变量
dcb. BaudRate=9600;                         //指定串口数据传输的波特率
    dcb.ByteSize =8;                        //设置串口数据位大小为 8 个字节

dcb.Parity = NOPARITY;                      //串口参数设置
dcb.StopBits = ONESTOPBIT;
dcb.fBinary = TRUE;
dcb.fParity = FALSE;
BOOL istrue;                                //定义布尔变量
istrue=SetCommState(hModem, &dcb);          //调用函数进行参数设置
if(istrue)                                  //判断串口参数是否设置成功
{
MessageBox("串口参数设置成功！");           //若参数设置成功，则提示用户成功
     }
else
{
MessageBox("串口参数设置失败！请重试");     //若参数设置失败，则提示用户重试
    }
COMMTIMEOUTS con;                           //定义结构体变量
con. ReadIntervalTimeout=10;                //设置串口数据读取的超时时间
BOOL istrue;                                //定义布尔变量
istrue= SetCommTimeouts(hModem, &con);      //调用函数进行参数设置
if(istrue)                                  //判断串口参数是否设置成功
{
MessageBox("超时时间设置成功！");           //若参数设置成功，则提示用户成功
     }
else
{
MessageBox("超时时间设置失败！请重试");     //若参数设置失败，则提示用户重试
    }
SetupComm(hModem,1024,512);                 //设置各数据缓冲区的大小
BOOL istrue;                                //定义布尔变量
istrue=PurgeComm(hModem,PURGE_TXABORT|PURGE_RXABORT|PURGE_TXCLEAR|
PURGE_RXCLEAR);                             //调用函数对缓冲区内容进行清除
if(istrue)                                  //判断清除是否成功
{
    MessageBox("缓冲区数据清除成功！");     //若参数设置成功，则提示用户成功
     }
else
{
    MessageBox("缓冲区数据清除失败！请重试");
                                            //若参数设置失败，则提示用户重试
    }
BOOL istrue;                                //定义布尔变量
istrue=SetCommMask(hModem, EV_RXCHAR | EV_TXEMPTY);
                                            //设置串口事件的类型
if(istrue)                                  //判断串口事件是否成功
{
    MessageBox("串口事件设置成功！");       //若事件设置成功，则提示用户成功
     }
else
{
```

```
    MessageBox("串口事件设置失败！请重试");
                                        //若事件设置失败，则提示用户重试
    }
BOOL istrue1;                           //定义布尔变量
DWORD de=0;                             //定义事件保存变量
}
```

从上面的代码中可以看到，在打开串口按钮的消息响应函数中，调用函数打开了串口并对该串口进行初始化。

2. 接收串口数据

实例程序的数据接收功能是在界面中接收数据按钮的消息响应函数 OnRecvdata()中实现的。其代码如下：

```
 void CMyView::OnRecvdata()
{
 istrue1= WaitCommEvent (hModem,&de,NULL);
                                   //等待事件的发生必须将第三个参数设置为NULL
if(istrue1)                        //判断事件等待是否成功
{
      MessageBox("串口事件等待成功！");//若事件等待成功，则提示用户成功
DWORD de=0;                        //定义事件保存变量
if(istrue)                         //判断事件等待是否成功
{
      if ((dwEvtMask== EV_RXCHAR)  //判断发生的串口事件是否为接收事件
      {                            //如果是，则表示缓冲区中有数据到达
           DWORD doe;              //定义清除的错误变量
            COMSTAT coms;          //定义结构体变量
             ClearCommError(hModem,&doe,&coms);
                                   //清除串口事件并获取当前状态
      if(coms. cbInQue>0)          //判断串口输入缓冲区中的数据大小
      {
      char buffer[coms. cbInQue]={0};  //定义并初始化缓冲区
        DWORD data;                //存放实际读取到的字节数
           BOOL f;                 //定义布尔变量
 f=ReadFile(hModem,&buffer, coms. cbInQue,data,NULL);
                                   //读取缓冲区中的数据
 if(!f)                            //判断读取函数的返回值
{
 BOOL m;                           //定义布尔变量
 m=GetOverlappedResult(hModem, NULL, data,true);
                                   //判断缓冲区中是否存在数据
 while(!m)                         //如果存在，则进入循环
{
 if (GetLastError() == ERROR_IO_PENDING)    //判断错误代码
 {
      m=GetOverlappedResult(hModem, NULL, data,true);
                                   //继续调用函数监视缓冲区数据
      }
           }
      }
    }
}}}
  CString str,str1;                //定义字符串
```

```
GetDlgItem(IDC_MSG)->GetWindowText(str); //获取消息接收框中原有的数据
str+="\r\n";                             //添加换行符
str+=str1;                               //连接数据
str+="\r\n";
GetDlgItem(IDC_MSG)->SetWindowText(str); //将数据显示在消息接收框中
}
```

当函数 WaitCommEvent()监视到有相应的串口事件发生时，该函数将返回 true。接着，用户便可以进行串口数据相关的读取操作了，如图 13.24 所示。



图 13.24　通过串口接收数据

3．发送串口数据

用户可以在发送数据按钮的消息响应函数 OnSenddata()中，实现串口数据的发送操作。代码如下：

```
 void CMyView::OnSenddata()                  //发送数据按钮消息响应函数
{
CString str,str1;                            //定义字符串变量
char* a;                                     //定义字符指针
DWORD data1;
GetDlgItem(IDC_EDIT2)->GetWindowText(str);   //获取发送编辑框中的内容
if(str.GetLength()!=0)                       //判断用户输入是否为空
{
BOOL f=WriteFile(hModem,str.GetBuffer(0), data,data1, NULL); //写入缓冲
                                                              区中的数据
if(!f)                                       //判断数据写入是否成功
{
   MessageBox("数据写入成功!");              //若数据写入成功，则提示用户成功
   }
else
{
    MessageBox("数据写入失败!");             //若数据写入失败，则提示用户失败
    }
```

```
  }
}
```

在发送数据按钮的消息响应函数中，用户首先获取发送编辑框中的内容。然后，判断用户输入的内容是否为 NULL。如果不为 NULL，则调用函数 WriteFile()将该数据写入串口。否则，向用户提示数据写入失败。

4．保存串口数据

在本章实例中，为了方便用户查看一些串口数据操作的相关记录。所以，用户需要将所操作过的串口数据写入文件中进行保存。实现数据保存是在界面中保存数据按钮的消息响应函数 OnSavedata()中进行的。代码如下：

```
 void CMyView::OnSavedata()
{
 CFile file("数据记录.txt",CFile::modeRead|CFile::modeWrite|CFile::
 modeCreate

                           |CFile::modeNoTruncate|CFile::typeBinary);
                                                //创建文件
 CString str;                                   //定义字符串变量
 char szdata[100];                              //数据缓冲区
 DWORD leth=file.GetLength();                   //获取数据写入前的文件长度
 GetDlgItem(IDC_MSG)->GetWindowText(str);       //获取消息接收框中原有的数据
 strcpy(szdata,str.GetBuffer(100));             //复制数据缓冲区中的数据
 file.SeekToEnd();                              //定位文件指针到最后
 file.Write("\r\n",2);                          //写入换行符
 file.Write(szdata,str.GetLength());            //写入数据
 file.Flush();                                  //强制写入
 if(file.GetLength()!=leth)                     //判断是否写入新数据
 {
        MessageBox("数据保存成功！");            //提示用户数据是否写入成功
}
else
{
    MessageBox("数据保存失败！");
}
 file.Close();
}
```

在程序中，用户首先获取数据保存文件的原始长度。然后，再将数据写入文件中保存并获取当前的文件长度。最后判断原始长度与当前长度的值。若该值不同，则认为写入数据成功。否则，认为写入数据失败，如图 13.25 所示。

5．关闭串口

用户在使用完串口或者应用程序退出时，都需要将打开的串口关闭。否则，该串口将一直处于打开状态，使其他程序或者该应用程序不能继续使用该串口。本实例中，将该功能实现在程序关闭消息 WM_CLOSE 的消息响应函数中。代码如下：

```
 void CMyView::OnClose()                 //关闭消息 WM_CLOSE 的消息响应函数
{
 BOOL is;                                //定义布尔变量
```

```
 is=::CloseHandle(hModem);                //调用函数关闭串口句柄
 if(is)                                   //判断串口是否关闭成功
 {
        MessageBox("串口成功关闭");        //提示信息
}
else
{
    MessageBox("串口关闭失败");
}
CFormView::OnClose();
}
```



图 13.25　数据保存成功的提示信息

本节通过应用 API 串口函数进行实例编程，向用户讲解各个串口编程相关的 API 函数的用法等。请用户参考随书光盘中相应章节的实例代码进行学习。

13.3　小　　结

本章通过实例程序各个功能的实现步骤，向用户讲解关于串口编程的相关知识。在实例程序中，分别通过使用 MFC 串口控件和串口 API 函数向用户介绍了这两种方法的使用步骤等。用户通过本章的学习可以学习到串口编程的一般流程以及相关函数和控件的使用方法等。用户学习完本章之后，应当能够独立进行串口实例程序的编写、调试等。

第 14 章　VC 发送手机短信

目前，由于在这个信息化的社会环境下，商业竞争变得越来越激烈。所以，越来越多的企业都在开发自己的短信平台，以求达到更快捷的信息资源或服务。一般情况下，用户都会选择使用 VC 平台与短信猫进行短信平台的开发。因此，在本章中，将向用户介绍短信猫相关的基本知识及其二次开发接口等相关内容。

14.1　短信猫介绍

用户在 VC 平台下开发短信平台时，短信猫是必不可少的硬件设施。所以，用户必须了解什么是短信猫及其种类。当然在开发时，作为程序员而言，最关心的问题还是短信猫的生产商所提供的二次开发接口。因此，在本节中，将向用户介绍短信猫的种类以及二次开发接口等相关知识。

14.1.1　短信猫简介

短信猫（GSM MODEM）是一种支持 GSM 无线通信的工业级调制解调器，其功能与用户日常所用的 MODEM（交换机）的功能基本一致。一般情况下，短信猫的核心部分是基于德国西门子的 GSM 模块的。用户只需插入国内移动通信运营商的 SIM 卡后，即可接入运营商的 GSM 网络中。这样，用户便可以通过短信猫，实现无线 GSM 通话、收发短信、数据等功能。

如果将短信猫与手机相比较，短信猫的核心模块与手机的核心模块一样。当短信猫接通电源以后，其内置软件便开始运行工作。如果用户将某个移动运营商的 SIM 卡插入到短信猫中。那么，短信猫便完全和手机一样，被接入到移动通信网络中进行工作。

同时，计算机可以通过串口或 USB 接口通过相应的连接线连接短信猫，通过一系列的 AT 指令，实现与短信猫的数据通信。例如，收发短信、拨打电话以及收发传真等。只是在本章实例中，仅需要使用其手发短信的功能即可。所以，用户可以不用其他的功能。

短信猫与手机最大的区别，在于手机有自带的屏幕、键盘和应用软件。而短信猫则需要用户根据其二次开发接口对其进行相关的驱动和控制。当前，用户在实际开发中所使用的短信猫，其外型结构有很多种，但是其核心技术都是一样的，如图 14.1 所示。

注意：图 14.1 中所示外型结构的短信猫是基于 USB 接口的一款短信猫。关于短信猫接口方面的知识将在 14.1.2 节中介绍。

14.1.2　短信猫分类

虽然短信猫的核心技术基本相同，但是根据计算机的接口和短信猫中接口模块的不同，短信猫也可以分为不同的种类。如果按照计算机的接口不同，短信猫可以分为串口短信猫、USB 接口短信猫等。如果按照短信猫中接口模块多少，短信猫可分为单口短信猫和短信猫池两种。因此，在本节中将向用户简单介绍这几种短信猫。

1．串口短信猫

串口短信猫是指该类短信猫与计算机之间的数据通信是通过串口进行传输的。其接口外型如图 14.2 所示。



图 14.1　常用短信猫外型结构



图 14.2　串口短信猫外型结构

当用户使用串口短信猫与计算机相结合，开发短消息平台时，开发人员可以通过计算机串口向短信猫发送 AT 指令完成数据通信等操作。

2．USB接口短信猫

USB 接口短信猫是指该类短信猫与计算机之间的数据通信是通过 USB 接口进行传输的。其接口外型如图 14.1 所示。

由于 USB 接口属于即插即用的计算机接口。所以，使用 USB 接口的短信猫时，其操作步骤非常简单。用户仅需要将短信猫插入计算机的 USB 接口即可实现数据通信，并且可以从计算机串口中取其相应的工作电压进行工作。所以，从价格上讲，USB 短信猫的市场价格也比较便宜。在这里，建议用户在实际开发时，应该使用 USB 接口的短信猫进行实例开发。

3．单口短信猫

单口短信猫是指在短信猫中，用户只能插入一张 SIM 卡，进行单个通道的数据通信，如图 14.3 所示。如果用户希望通过不同的通道发送和接收多个数据，那么应该采用多口的短信猫进行开发。

4. 短信猫池

短信猫池是指该类短信猫具有多个通道，可以插入多张 SIM 卡，并且能够同时发送和接收多个数据，如图 14.4 所示。



图 14.3　单口短信猫外型结构



图 14.4　短信猫池外型结构

如图 14.4 所示，短信猫池具有多个数据传输通道，可以插入多张 SIM 卡，并且每个通道都具有各自的数据传输天线。如果用户开发的短消息平台需要以不同的号码群发短消息，那么应该使用该类型的短信猫进行平台开发。

在本节中，主要向用户介绍了短信猫的几种类型以及外型结构。通过本节知识的学习，用户对短信猫的种类以及类型应该有一个系统的了解。

14.1.3　短信猫开发接口

短信猫开发接口（GSM MODEM SDK）是指程序员编程与短信猫进行数据通信时，短信猫的生产商为程序员提供的一系列函数或者控件等。一般情况下，短信猫的生产商为程序员提供了 4 种开发接口模式。这 4 种开发接口模式分别为使用 AT 指令、短信猫二次开发包、短信猫通信中间件以及第三方提供的短信网关。在本节中，将向用户分别介绍这 4 种开发接口模式。

1. 使用AT指令

AT 指令是指一种基于调制解调器的命令语言。一般情况下，该指令是从一个终端设备或者是数据终端设备向终端适配器、数据电路终端设备发送的指令。通过向终端设备发送 AT 指令可以实现控制其功能的作用。例如，当用户需要获取插入短信猫中的 SIM 的相关信息时，便可以使用 AT 指令实现。其指令代码如下：

```
AT+CCID                                       //获取短信猫中的 SIM 卡相关信息
```

在 AT 指令中，均以字符 AT 作为指令开始。上面中的指令 AT+CCID 表示用户将使用该指令获取短信猫中插入的 SIM 卡的标识，而这个命令将使短信猫中相应的模块读取 SIM

卡上的 EF-CCID 标识文件。

注意：在这里仅仅是为了向用户介绍 AT 指令的作用以及基本格式。关于该指令的详细讲解将在 14.3 节中向用户进行讲解。

2．短信猫二次开发包

短信猫二次开发包是指短信猫的生产商通过对 AT 指令进行底层封装后，提供给上层开发人员的 API 函数或者控件等。

当用户需要使用短信猫中相应的功能时，只需要调用生产商所提供的短信猫二次开发包中的相关 API 函数等即可。虽然短信猫的生产商封装了底层的 AT 指令，但是用户只要对 AT 指令非常了解，也可以实现自行封装 AT 指令而构造短信猫的二次开发包。例如，用户将获取短信猫中插入的 SIM 卡的相关信息的 AT 指令封装为一个函数。代码如下：

```
void GetSIM()                                   //封装的 AT 指令函数
{
    ...                                         //省略部分代码
    char data[]={"AT+CCID"};                    //定义 AT 指令字符数组
    DWORD data1;                                //定义变量保存实际写入的指令大小
    BOOL istrue;                                //确定指令发送是否成功
istrue=::WriteFile(handle,&data,sizeof(data),data1,NULL);
                                                //将 AT 指令字符通过串口进行发送
      if(istrue)                                //判断 AT 指令是否发送成功
{
      MessageBox("获取 SIM 卡信息成功！");       //提示用户发送结果
}
else
{
      MessageBox("获取 SIM 卡信息失败！");
 }
}
```

在上面的代码中，用户可以看到像短信猫发送 AT 命令是通过串口进行传输的，函数 WriteFile()的第一个参数 handle 表示串口的句柄。但是，该种发送 AT 指令的方法仅适合于串口型的短信猫。

如果用户使用的短信猫为 USB 接口类型，则需要 RS-232 串口转 USB 接口的转换器也可实现通过串口发送 AT 指令。

3．短信猫通信中间件

短信猫通信中间件是指一套专门的针对数据库接口的短信猫通信软件。用户使用该类型的通信中间件，仅需提交短信队列到数据库即可进行短信收发。因此，无论用户所使用的是哪一种开发语言进行短信猫的二次开发，只需要对其数据库进行读写即可。这种开发简单快速，节约开发成本，是目前最为快捷的一种短信应用开发模式。

4．使用短信网关

短信网关是指由第三方开发的应用程序或提供的程序开发接口。一般情况下，这类短

信网关都是基于网页提供给用户使用的。通常，第三方首先将短信操作平台的相关功能集成到网页中。然后，用户便可以使用其提供的网页地址，将相关的数据转换为变量通过该网页地址传送到网页中的相关参数中。例如，用户假设一个第三方所提供的网页地址为“http://www.smsgate.cn/basic.asp”，而用户将发送的短信相关数据等以参数的形式通过该网址进行传递，则最终构造的网址如下：

```
http://www.smsgate.cn/basic.asp?mob=13778745236&pwd=mypassword&tel=
08256662151;13558979637&msg=短信内容
```

用户在实际编程时，只需要按照参数的一定顺序构造好该网址以后，便可以将其打开实现短消息的发送。在构造的网址中，其参数及意义如下：

- 参数 mob 表示用户在第三方处注册的电话号码。该电话号码是用户为了使用第三方所提供的短信网关而注册的，相当于用户名。
- 参数 pwd 表示用户注册时，所填写的密码。
- 参数 tel 表示接收短消息的电话号码。如果接收方为多个电话号码，则将各个号码之间使用符合“;”隔开即可
- 参数 msg 表示短消息的相关内容。

注意：当用户构造该网址时，必须将各个参数及其参数值之间使用符合“&”进行连接。

例如，用户使用 VB 程序将短消息中的相关内容构造为第三方所提供的短信平台网址。代码如下：

```
Function geturl(ByVal strParameter As String) As String
Dim s As String                            //定义字符串变量 s
Dim i As Integer                           //定义 int 型变量
Dim intValue As Integer                    //定义 int 型变量
Dim TmpData() As Byte                      //定义字节变量
s=""                                       //初始化字符串
TmpData=StrConv(strParameter, vbFromUnicode)//调用系统函数
For i=0 To UBound(TmpData)                 //For 循环
intValue=TmpData(i)                        //赋值
If (intValue >= 48 And intValue <= 57) Or (intValue >= 65 And intValue <=
90) Or (intValue >= 97
                                           //判断数据的范围以及识别
And intValue <= 122) Then
s=s & Chr(intValue)                        //连接字符串
ElseIf intValue=32 Then                    //判断字符
s=s & "+"                                  //连接符号"+"
Else
s=s & "%" & Hex(intValue)                  //将值转换为十六进制并连接
End If
Next i
geturl=s                                   //返回构造好的网址
End Function                               //结束函数
```

如果用户使用第三方所提供的短信网关。那么，用户在开发短信平台时，不需要再使用短信猫等相关的硬件设备了。而仅仅需要将短消息的相关内容进行组织以后，构造成第三方所规定的网址后，将其打开即可。这种方法使用简单，易于实现。但是，其局限性非

常大，用户会受到第三方的一些约束等。

在本节中，主要向用户介绍了关于短信猫二次开发接口相关的几种方法。并且介绍了这几种方法的优点以及缺点等。通过本节的学习，用户将了解对短信猫二次开发接口方面的知识。

14.2　实现与短信猫的硬件连接

用户使用短信猫时，应该首先确保 PC 与短信猫之间的硬件连接无误后，方可进行相关的操作。所以，在本节中，将向用户介绍短信猫相关的硬件设备以及实现 PC 与短信猫的硬件连接方法。

14.2.1　短信猫的硬件设备

一般情况下，短信猫的硬件设备较为简单，主要由几部分组成。在本节中，将向用户介绍这些硬件设备的外型结构以及作用等。

1．短信猫主机

首先，用户应该获得短信猫的主机，这是硬件中最重要的一部分。由于短信猫有两种接口模式。所以，用户可以根据需要选择合适的短信猫主机。例如，用户选择 USB 接口的短信猫作为短信猫主机，如图 14.5 所示。

在图 14.5 中，所示的短信猫是 USB 接口模式的。如果用户需要使用串口模式的短信猫，则其外型结构如图 14.2 所示。一般情况下，用户选择 USB 接口的短信猫可以节约成本，缩短开发周期等。

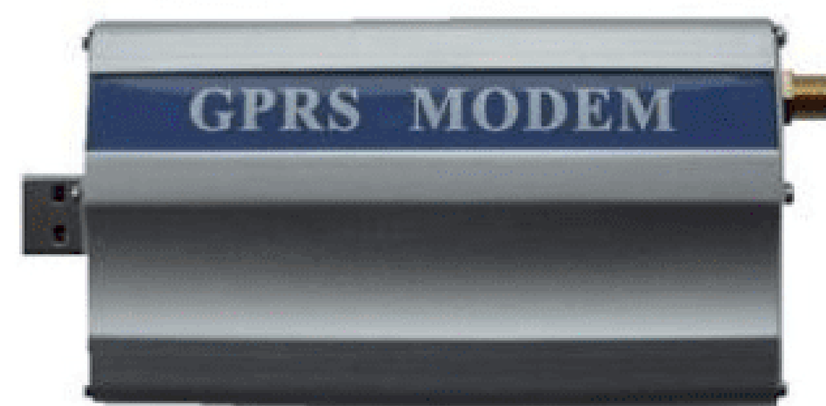


图 14.5　USB 接口短信猫

2．电源线与数据传输线

在短信猫与计算机之间需要一根数据线连接，才能实现数据通信。例如，用户使用的短信猫是 USB 接口，则数据线应该选择一根 USB 接口线。如果用户使用的短信猫是串口接口，则数据线选择一根串口线即可。

通常，USB 接口的短信猫可以从计算机上获得电源进行工作。所以，用户使用 USB 接口的短信猫时，是不需要另外使用单独的电源线为其供电的。但是，串口模式的短信猫就需要用户单独配上相应的电源才能工作。

3．天线

由于短信猫工作时，是无线传输数据信号的。所以，用户使用短信猫时，还需要为其配置相应的天线，如图 14.6 所示。

注意：短信猫的天线可以用来接收或者发送用户需要的数据等。当然，图 14.6 中所示的天线为一般插接式天线。该天线最大的缺点是安装过程较繁杂。

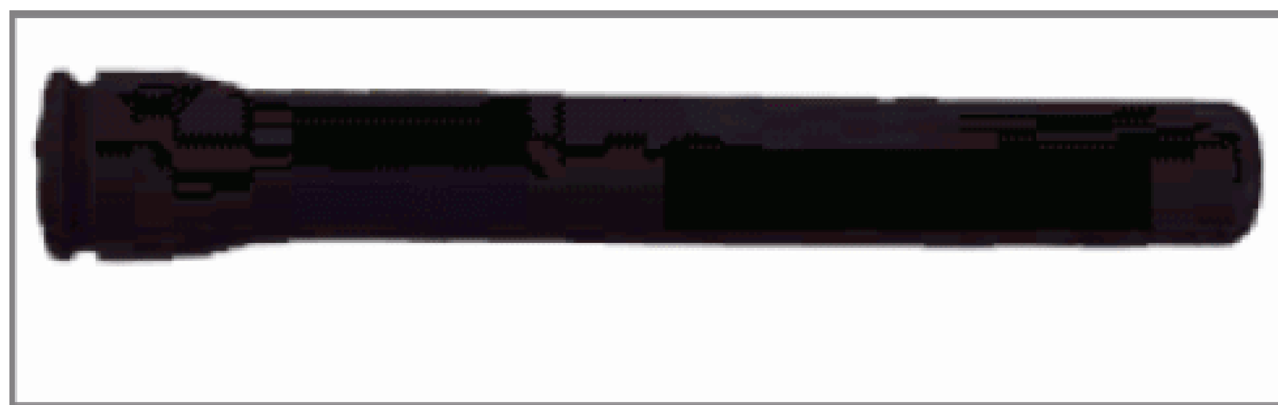


图 14.6　短信猫天线

实际上，短信猫的天线还有一种吸盘式的天线。安装这种天线比较方便并且快速。当用户使用时，将其吸盘放置在短信猫上即可。其大致外型结构如图 14.7 所示。



图 14.7　短信猫吸盘天线外型结构

上面所讲的硬件设备基本上就是短信猫的所有设备了。但是，用户进行二次开发还需要短信猫生产商所提供的短信猫二次开发包。这个二次开发包对于用户而言是非常重要的。

在本节中，主要向用户介绍了短信猫的硬件设备及其外型结构和基本作用等。

14.2.2　实现 PC 与短信猫连接

现在，用户已经可以使用短信猫相关的硬件与计算机相连接，实现数据交换了。这个功能需要经历两个步骤才能完成。这两个步骤分别为短信猫本身的硬件连接以及计算机配置。在本节中，将向用户分别介绍这两个步骤的实现方法。

1. 短信猫硬件连接

首先，用户需要将短信猫的电源断掉，并弹出 SIM 卡座，将 SIM 卡放入该卡座中。用户在插入过程中，应将 SIM 卡的金属面朝上，并且应该十分小心，以免弄坏 SIM 卡。如果 SIM 卡成功放入卡座中，用户应该将卡座轻轻收回。

然后，用户需要将短信猫天线安装到短信猫上，如图 14.8 所示。短信猫天线安装成功后，用户便可以使用串口线或者 USB 连接线将短信猫和计算机相应的接口连接起来，如图 14.9 所示。

连接成功后，用户需要为短信猫插上电源，使其进入工作状态。如果用户使用的短信猫是基于 USB 接口，那么该短信猫会直接从计算机中取电运行。

用户需要等待短信猫运行 2 到 3 分钟后，查看其工作指示灯是否点亮。如果该指示灯没有被点亮，则短信猫硬件连接出现问题，需要重新进行检查。否则，表示连接正确，短

信猫已经进入工作状态。

2．计算机配置

用户待短信猫的硬件安装成功后，还需要对计算机进行相应的配置才可以实现计算机与短信猫进行数据通信。用户在计算机中，需要添加的服务分别为添加调制解调器以及添加相应的网络连接。首先，用户为计算机添加调制解调器是为了校准对应的串口。其步骤如下：



图 14.8　短信猫安装天线后的外型



图 14.9　使用连接线连接短信猫和计算机

（1）用户选择“开始”|“控制面板”命令，打开控制面板，如图 14.10 所示。

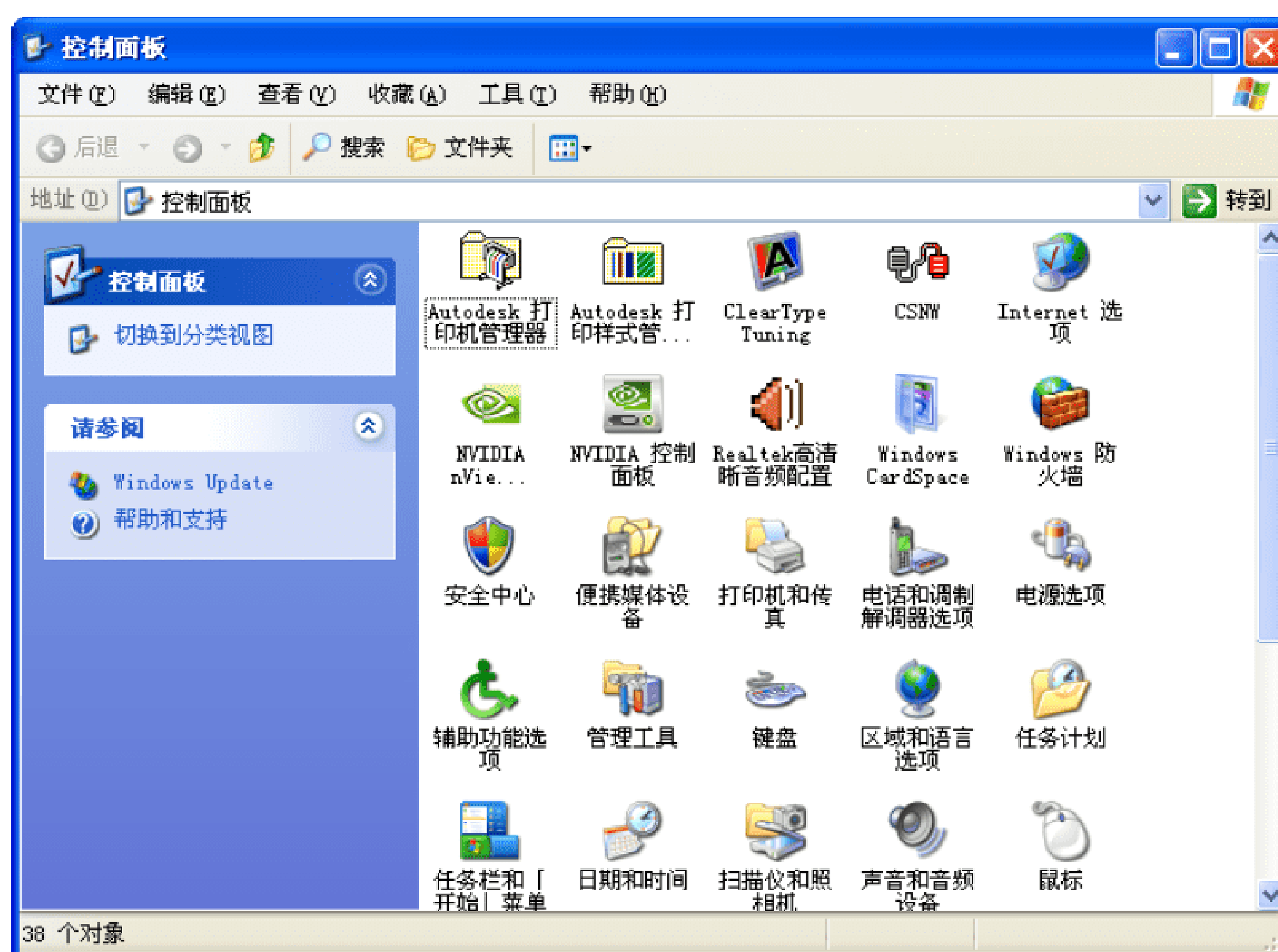


图 14.10　控制面板

（2）用户单击控制面板中相应的文件列表项“电话和调制解调器选项”并打开该选项

对话框，如图 14.11 所示。

🔔注意：如果用户的计算机中已经连接了调制解调器，那么在图 14.11 所示的对话框中，会显示相应的调制解调器名称以及串口号。

（3）单击“添加”按钮，打开“添加硬件向导”对话框，如图 14.12 所示。

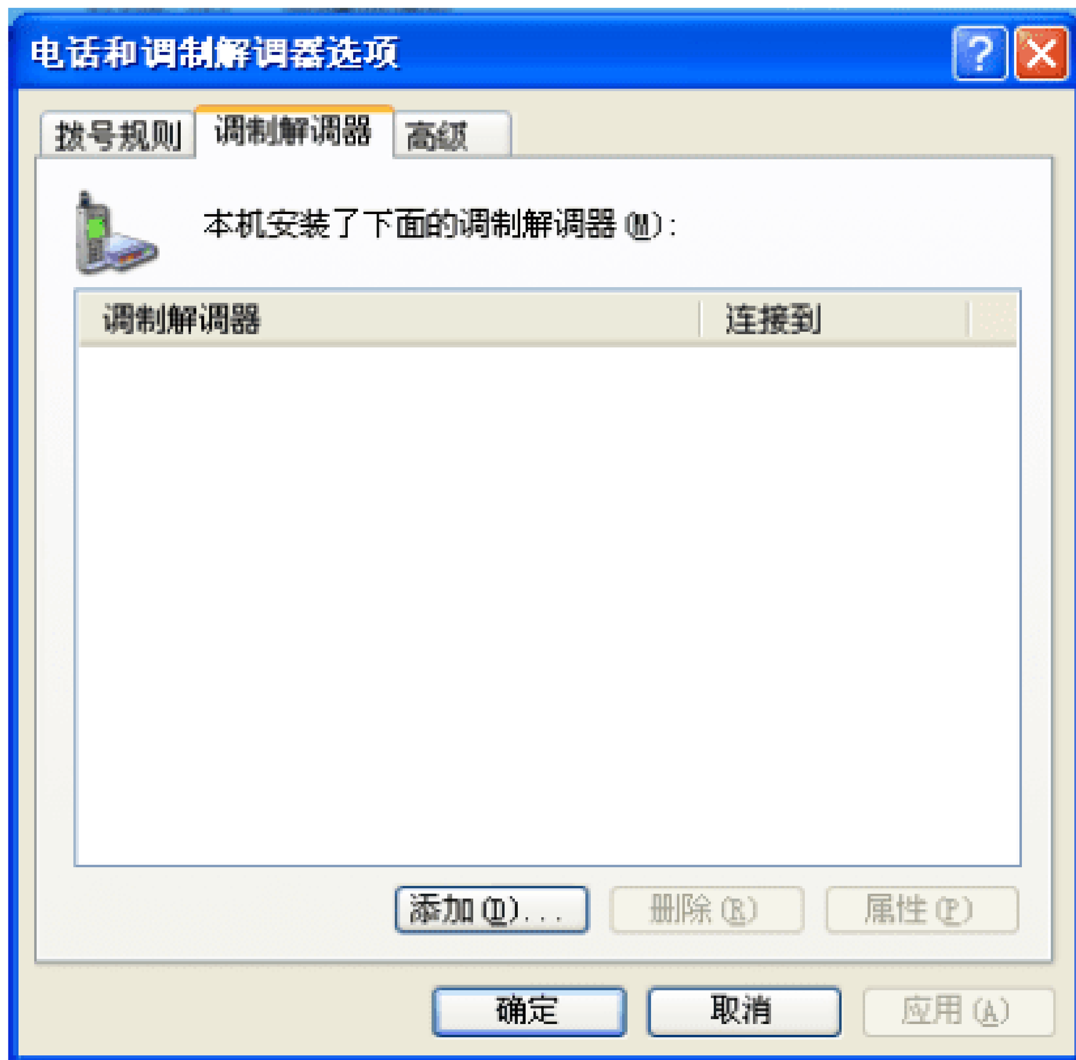


图 14.11　“电话和调制解调器选项”对话框

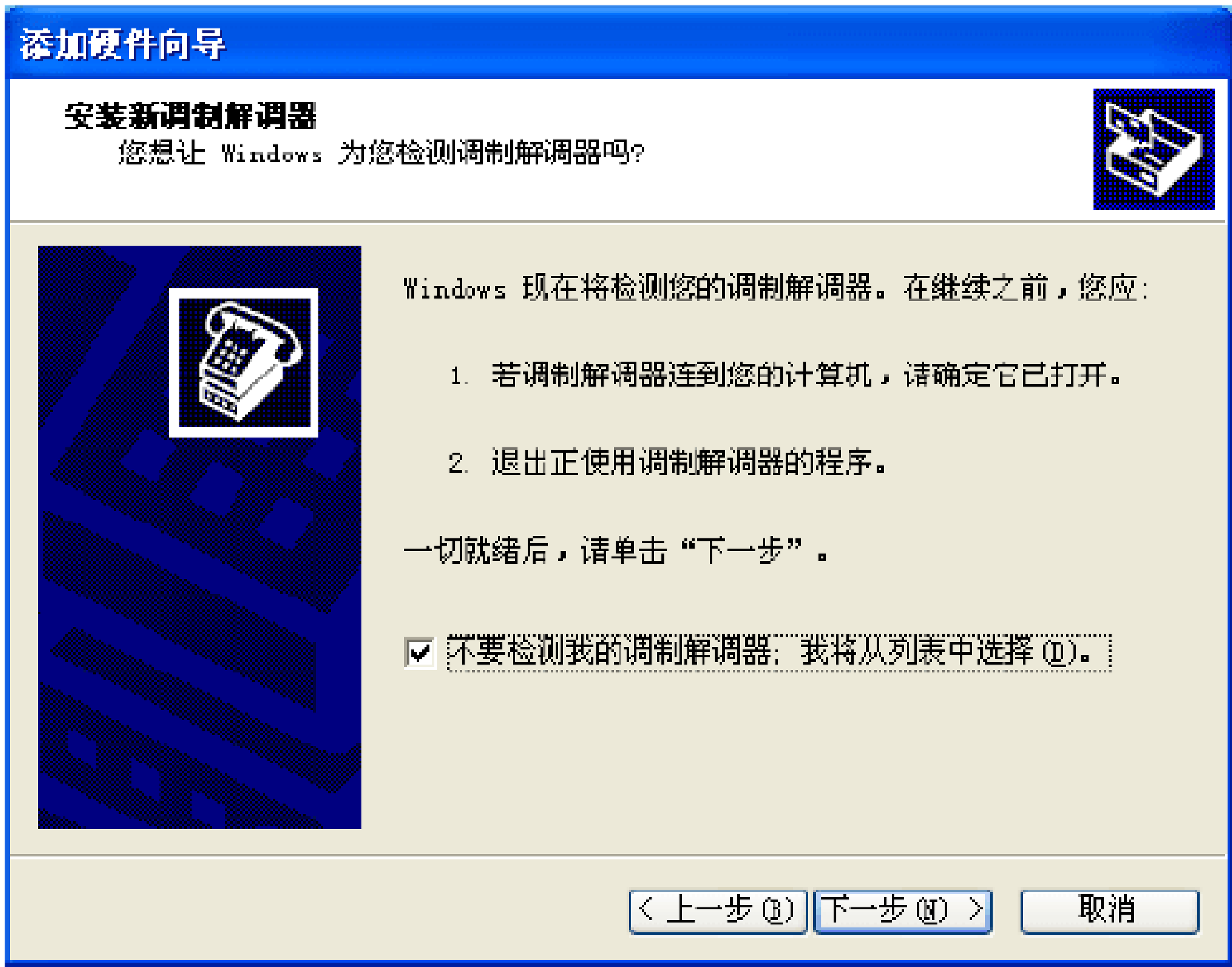


图 14.12　“添加硬件向导”对话框

在该向导对话框中，如果用户选择“不要检测我的调制解调器材：我将从列表中选择”复选框，表示向导程序将不执行检测。一般情况下，用户均需要选择该复选框。

（4）单击“下一步”按钮，开始选择并安装调制解调器，如图 14.13 所示。

（5）选择调制解调器以后，单击“下一步”按钮，选择串口号，如图 14.14 所示。



图 14.13　选择调制解调器



图 14.14　选择串口号

（6）单击“下一步”按钮完成串口的选择并成功安装调制解调器，如图 14.15 所示。

（7）用户单击“完成”按钮便成功为计算机添加并安装了调制解调器。此时，用户便可以查看刚安装的调制解调器的相关信息了，如图 14.16 所示。

在本节中，向用户介绍了如何为计算机添加调制解调器的相关配置方法。通过本节的学习，用户也可以使用同样的方法为计算机添加其他硬件设备等。

3．添加网络连接

用户要实现计算机与短信猫的通信，还需要为计算机添加相应的网络连接，才可以成功实现通信。其基本的操作步骤如下：

（1）用户可以右击桌面上的“网上邻居”，在弹出的快捷菜单中选择“属性”命令，将弹出“网络连接”对话框，如图 14.17 所示。



图 14.15　完成调制解调器的安装

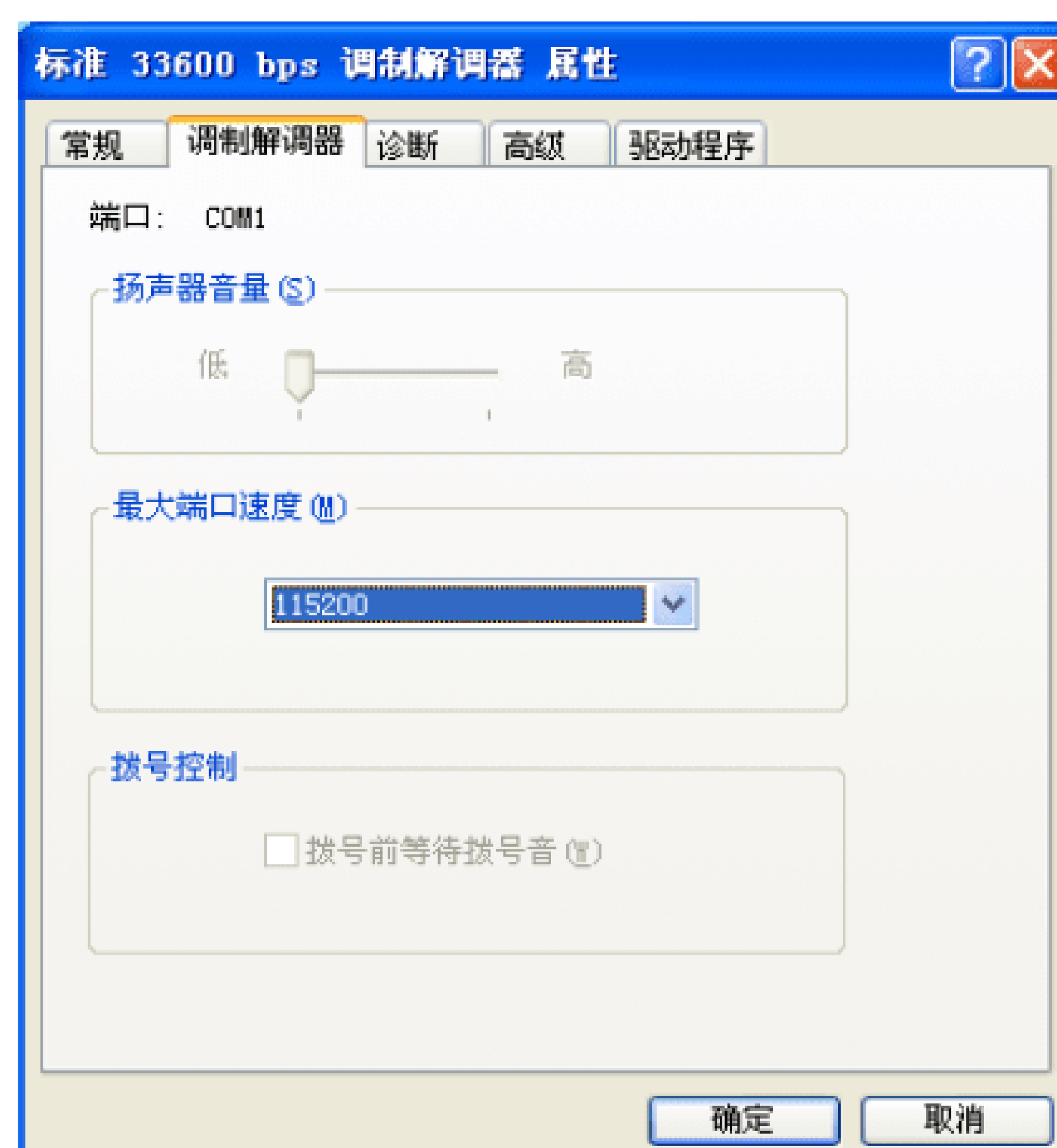


图 14.16　查看调制解调器的相关信息

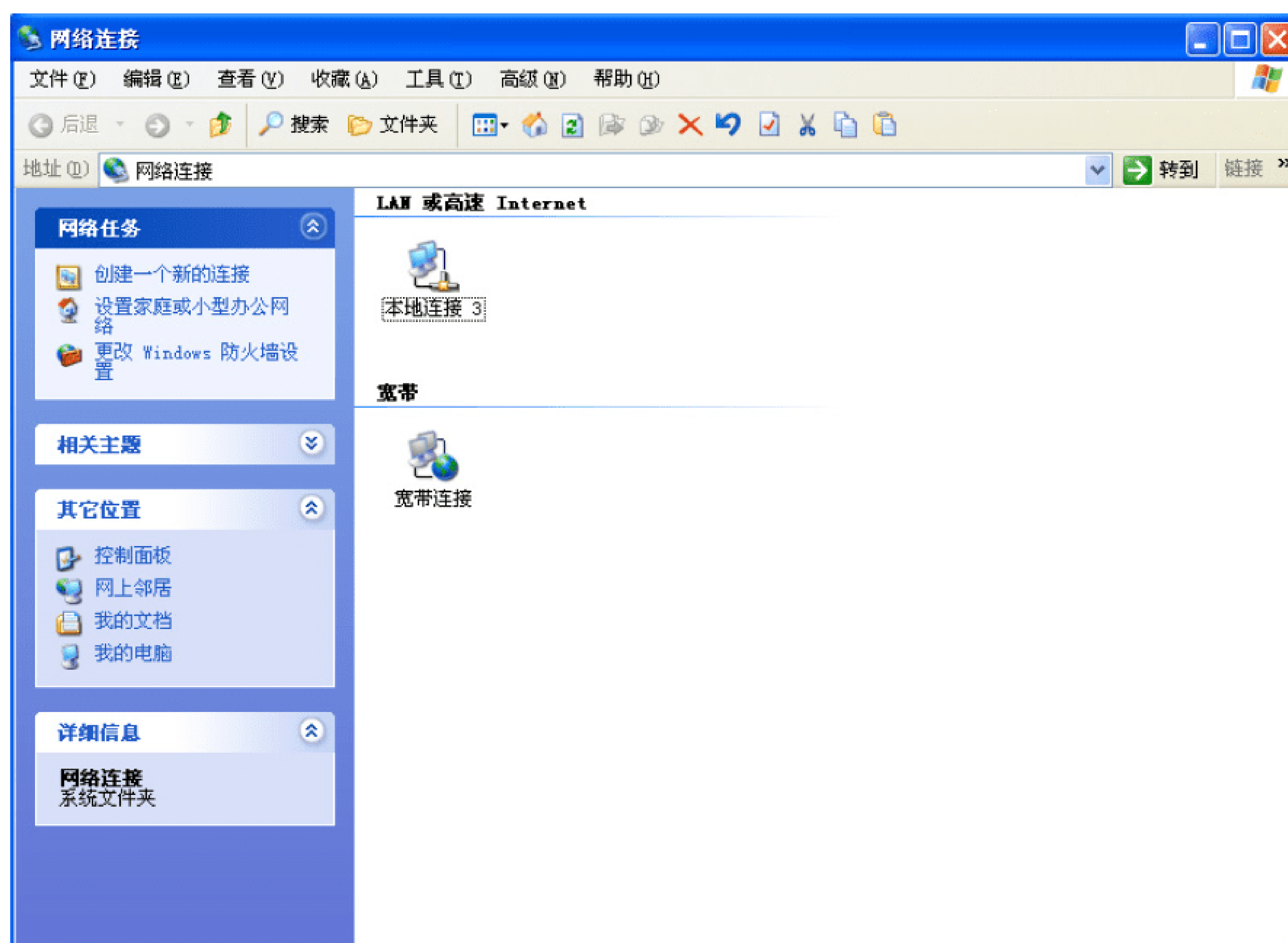


图 14.17　“网络连接”对话框

🔔注意：用户除了使用本章中所介绍的方法打开“网络连接”对话框以外，还可以使用操作系统中的控制面板打开。

（2）用户在“网络任务”栏中，选择“创建一个新的连接”选项，将弹出“新建连接向导”对话框，如图 14.18 所示。

（3）单击“下一步”按钮，选择网络连接类型，如图 14.19 所示。

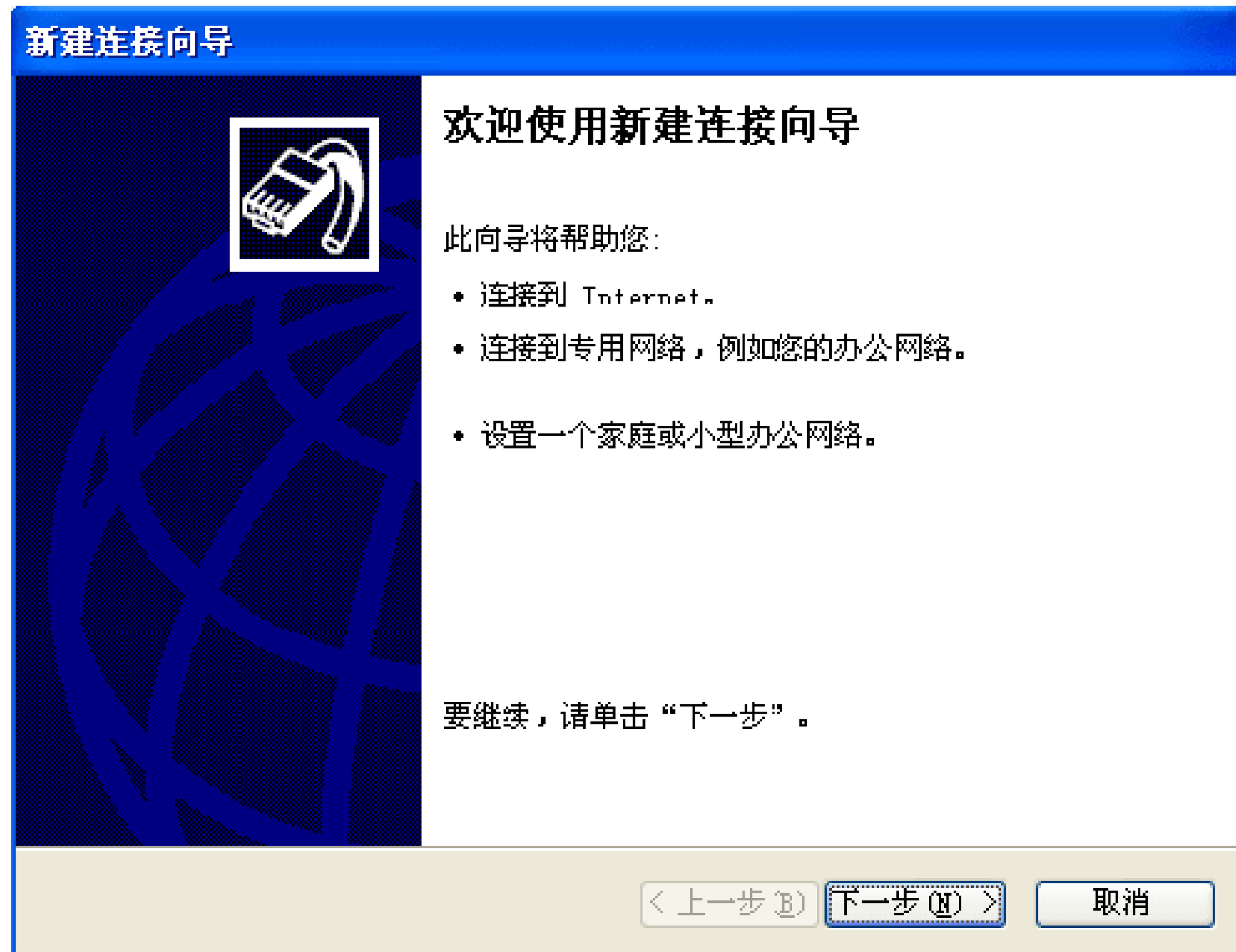


图 14.18 “新建连接向导”对话框

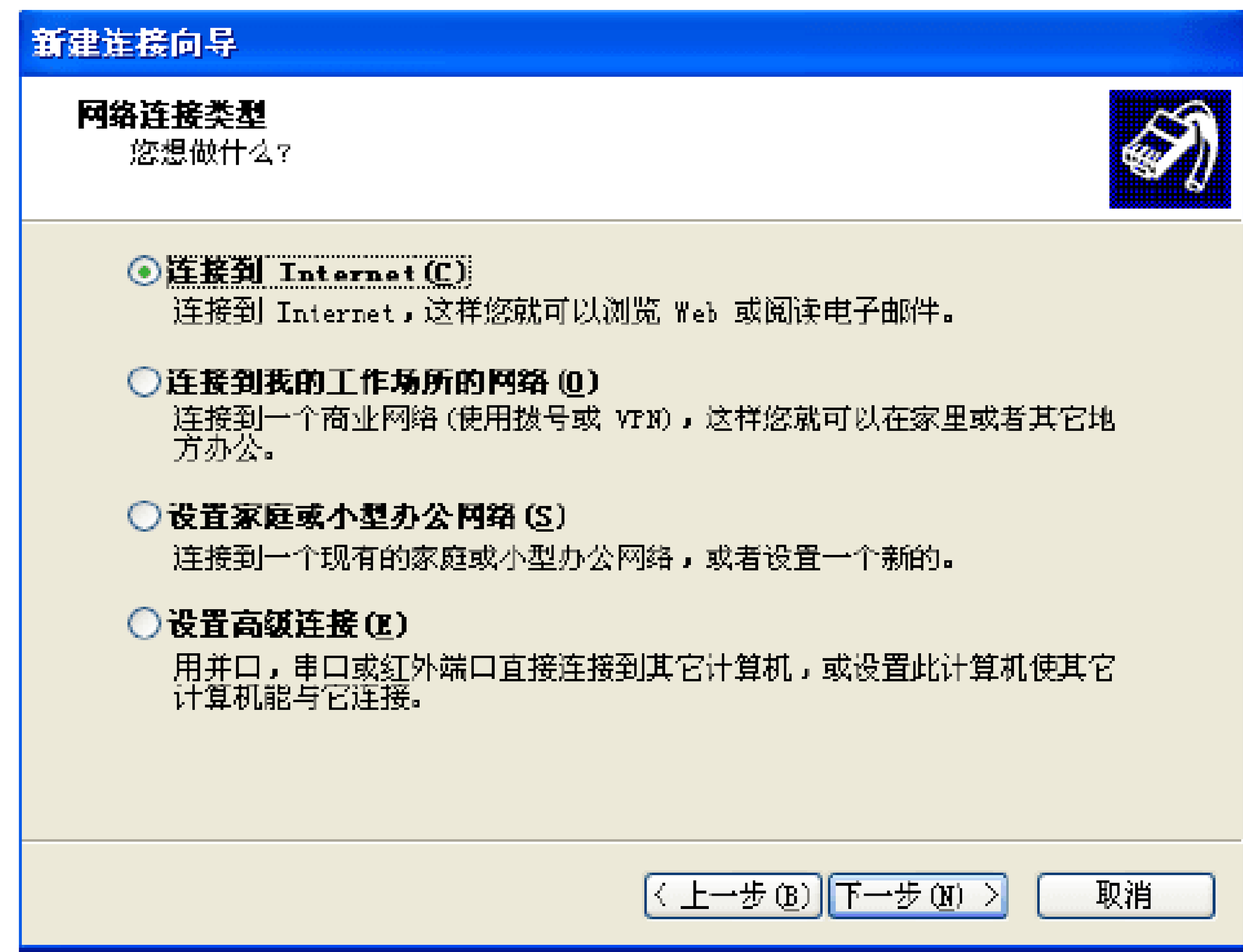


图 14.19 选择网络连接类型

🔔注意：在这一步中，用户需要选择连接的网络类型为“连接到 Internet”。

（4）单击“下一步”按钮，选择“手动设置我的连接”单选按钮，如图 14.20 所示。

（5）单击“下一步”按钮，选择“用拨号调制解调器连接”选项，如图 14.21 所示。

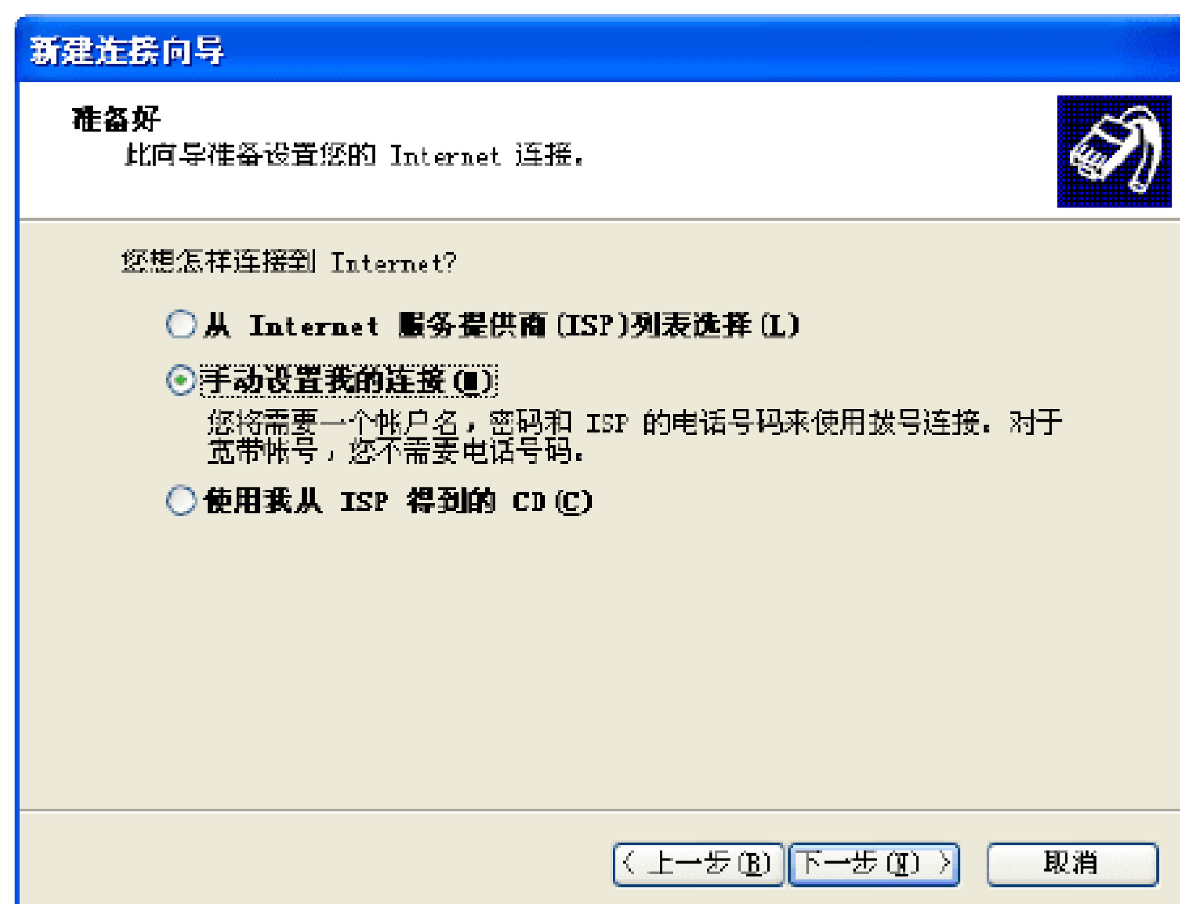


图 14.20 选择网络连接方式

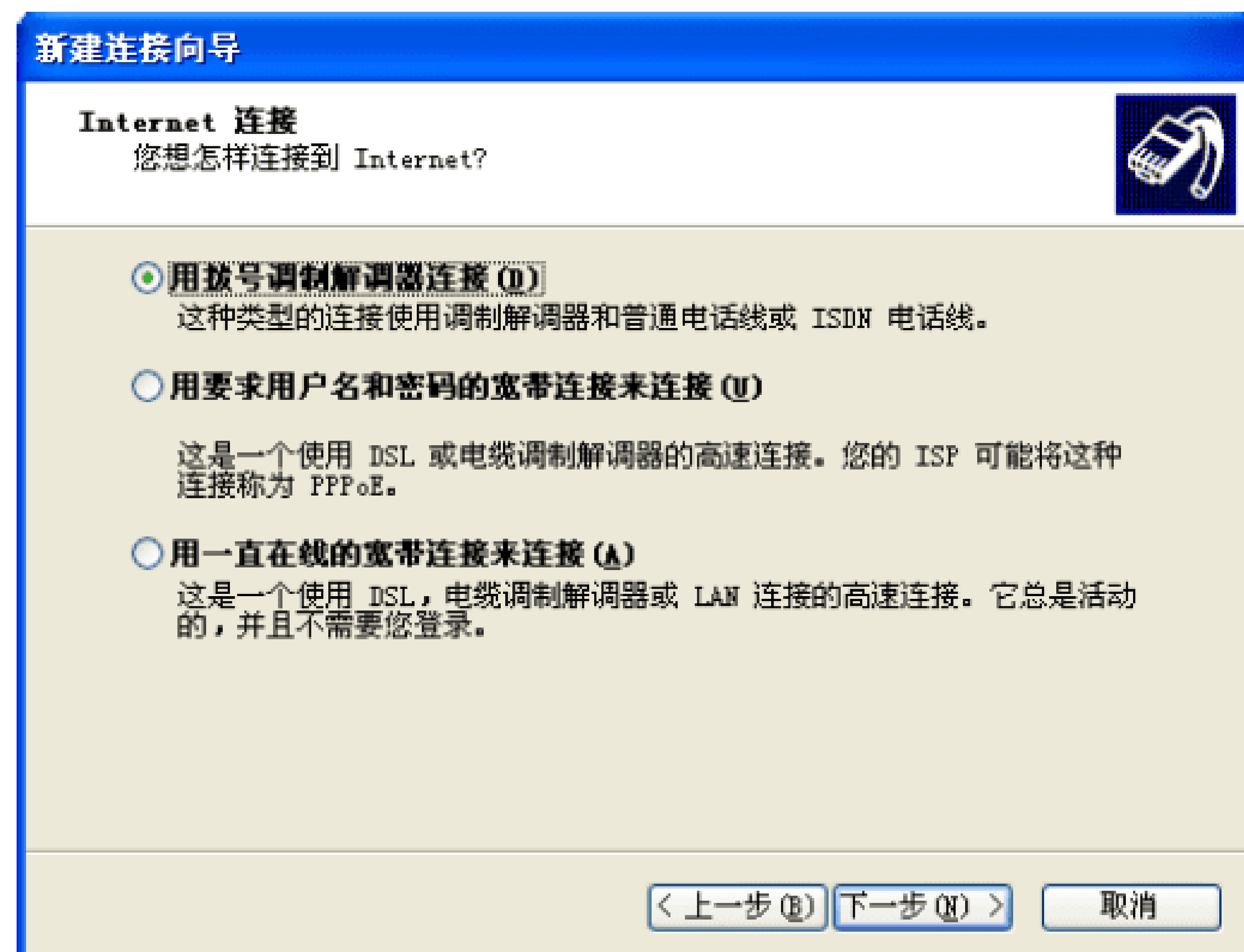


图 14.21 选择使用拨号调制解调器连接

（6）单击“下一步”按钮，修改服务提供商的名称为 GPRS，如图 14.22 所示。

（7）单击“下一步”按钮，修改服务提供商的电话号码。由于在这里用户所使用的服务提供商是调制解调器。所以，将该电话号码修改为“*99***1#”，如图 14.23 所示。

（8）单击“下一步”按钮，修改用户的账户信息，如图 14.24 所示。

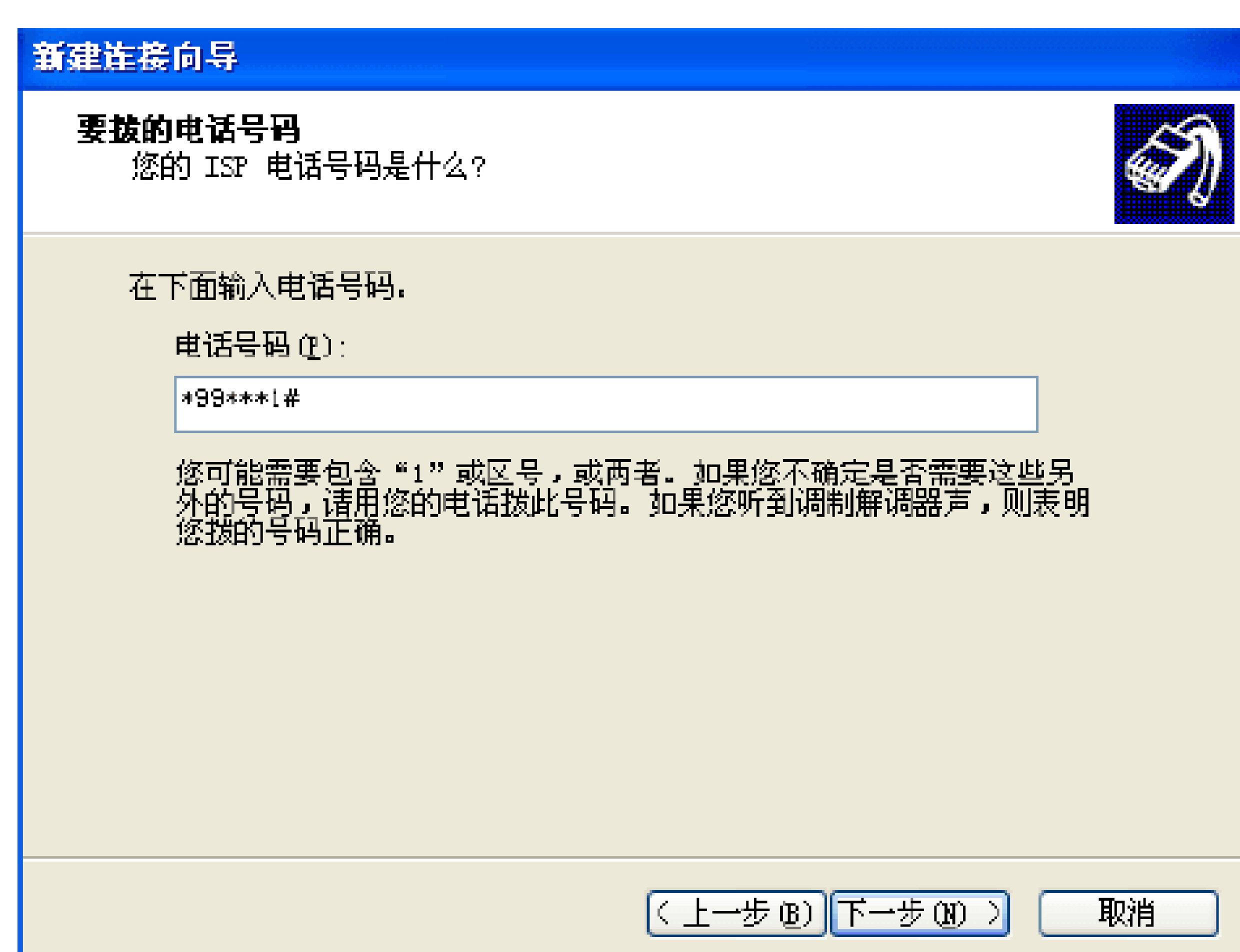


图 14.22　修改服务提供商的名称

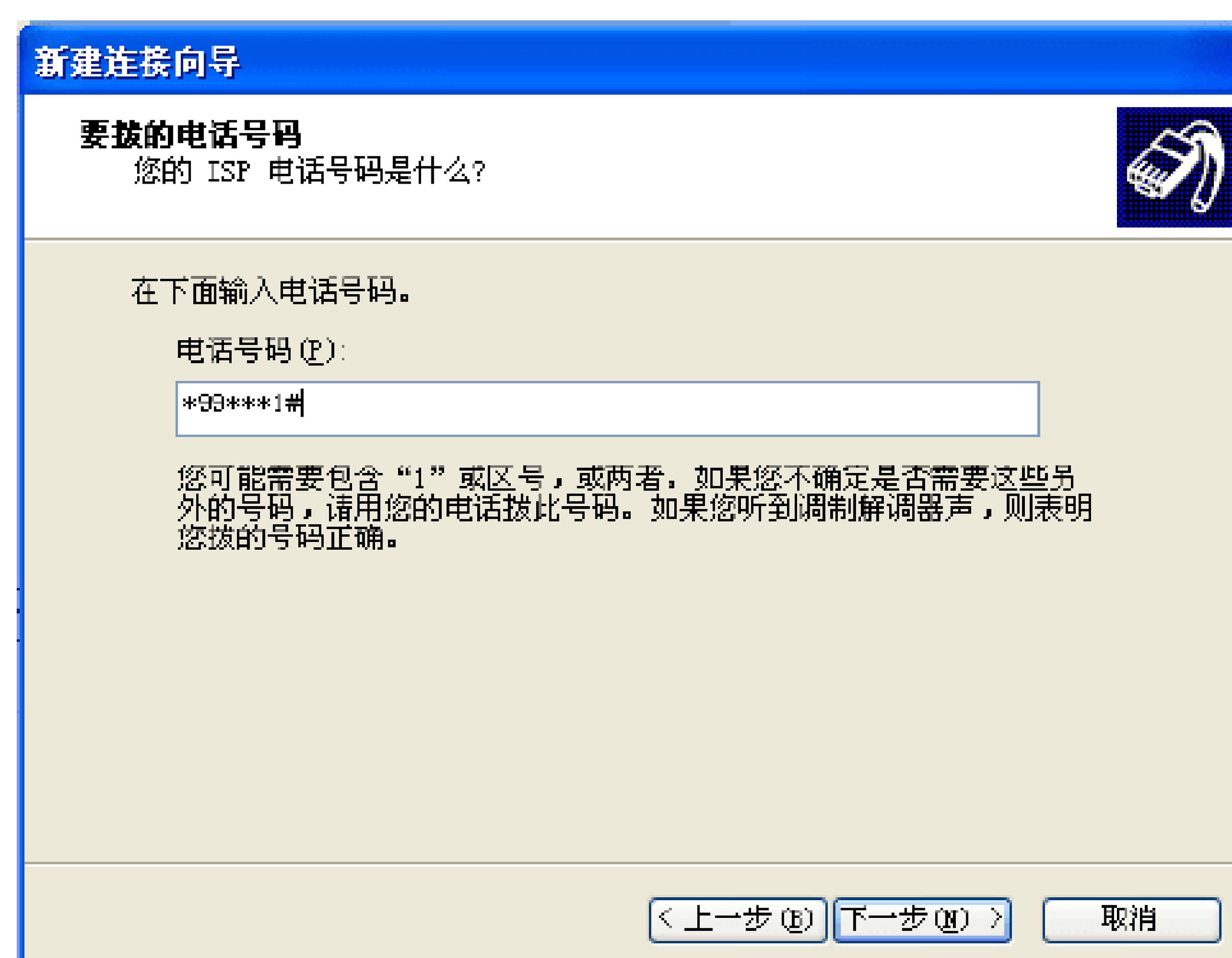


图 14.23　修改服务提供商的电话号码

注意：用户在这一步修改账户信息时，可以将这些信息保留为空，表示默认。

（9）单击“下一步”按钮，进入创建网络连接的最后一步，选择是否在桌面上添加一个快捷方式，如图 14.25 所示。

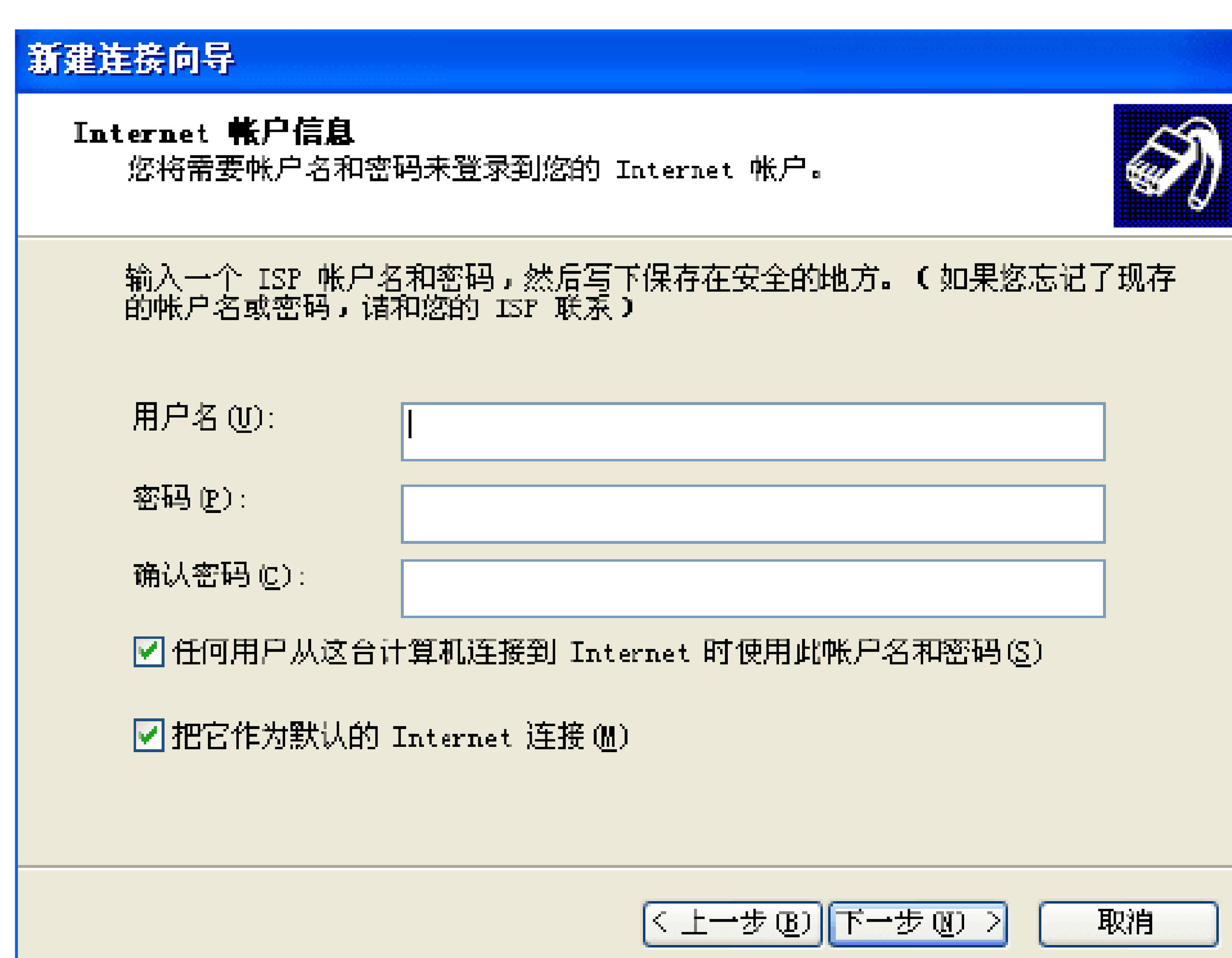


图 14.24　修改用户的账户信息

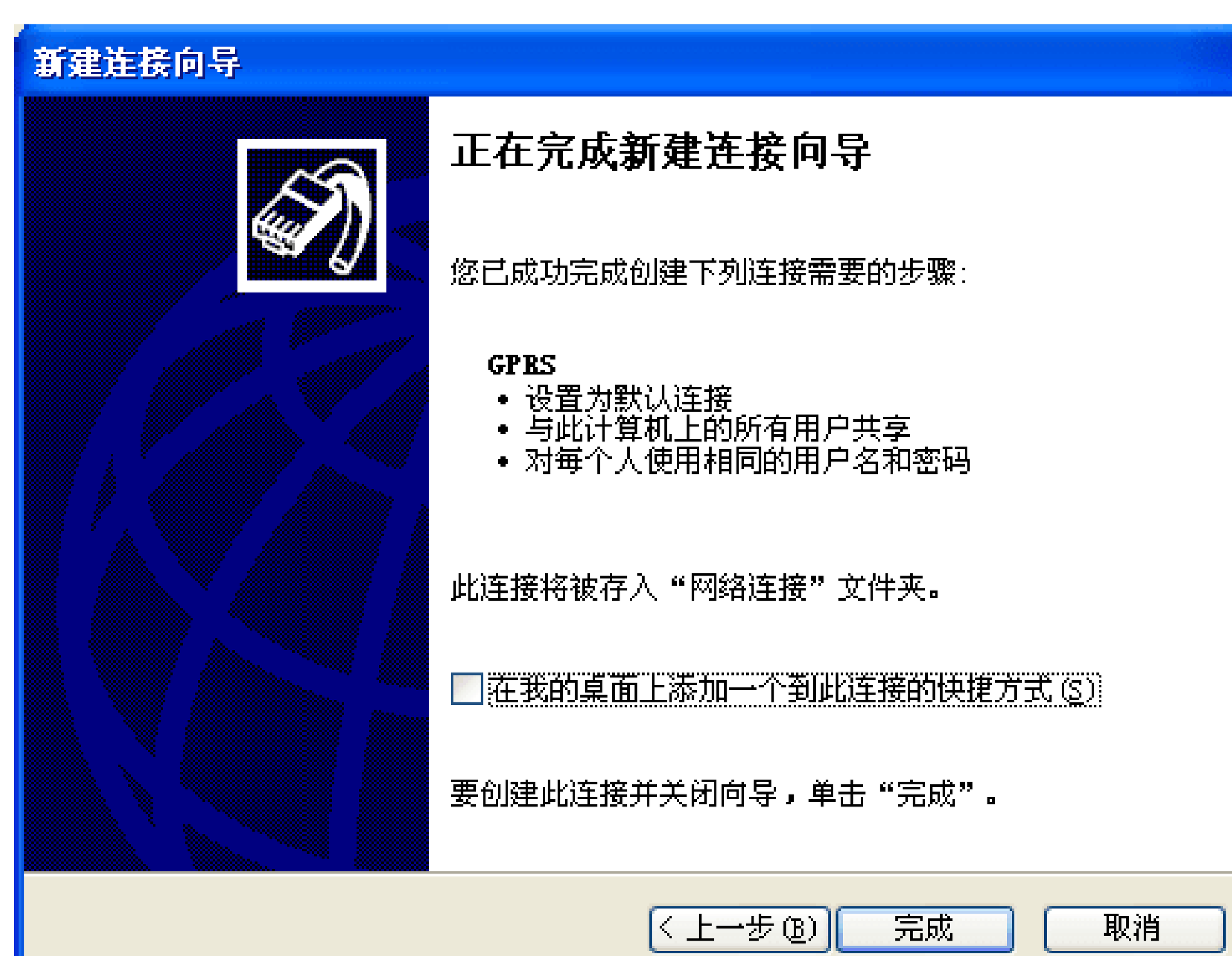


图 14.25　完成创建网络连接的最后一步

用户在最后一步中，可以直接单击“完成”按钮，完成网络连接的创建。如果用户在前面的设置中有错误，也可以单击“上一步”按钮，到达相应的步骤中修改相关设置。

在本节中，主要向用户介绍了创建一个网络连接的方法，并介绍了其中的一些相关设置信息等。

14.3　相关 AT 指令介绍

本书在前面的小节中，已经向用户讲解了短信猫的相关硬件以及这些硬件的连接等。用户真正操作短信猫，还需要向其发送相应的指令，这些指令称为“AT 指令”。在本节中，将向用户讲解短信猫中相应功能的 AT 指令代码。

14.3.1　AT 指令介绍

AT 指令是指计算机向其附加的硬件设备发送的相关功能命令，或者是计算机所带的硬件。例如，硬盘读写操作命令等。

通常情况下，用户可以通过选择“开始”|“运行”命令，打开计算机中的“运行”对话框，并在文本框中输入 cmd，单击“确定”按钮，打开命令运行对话框，如图 14.26 所示。

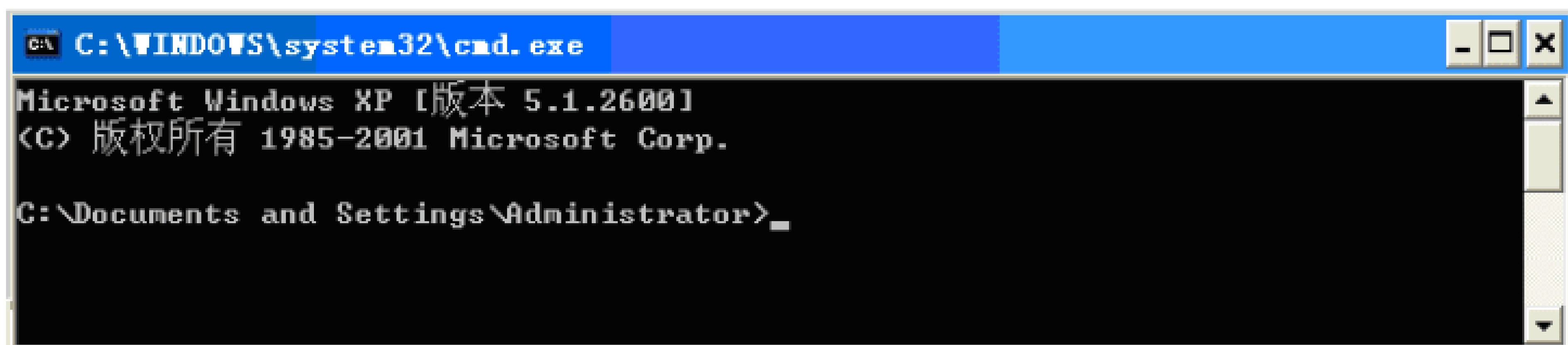


图 14.26　命令运行对话框

用户在该窗口的光标处输入“AT+空格+R”，即可阅读 AT 指令的相关帮助信息，如图 14.27 所示。

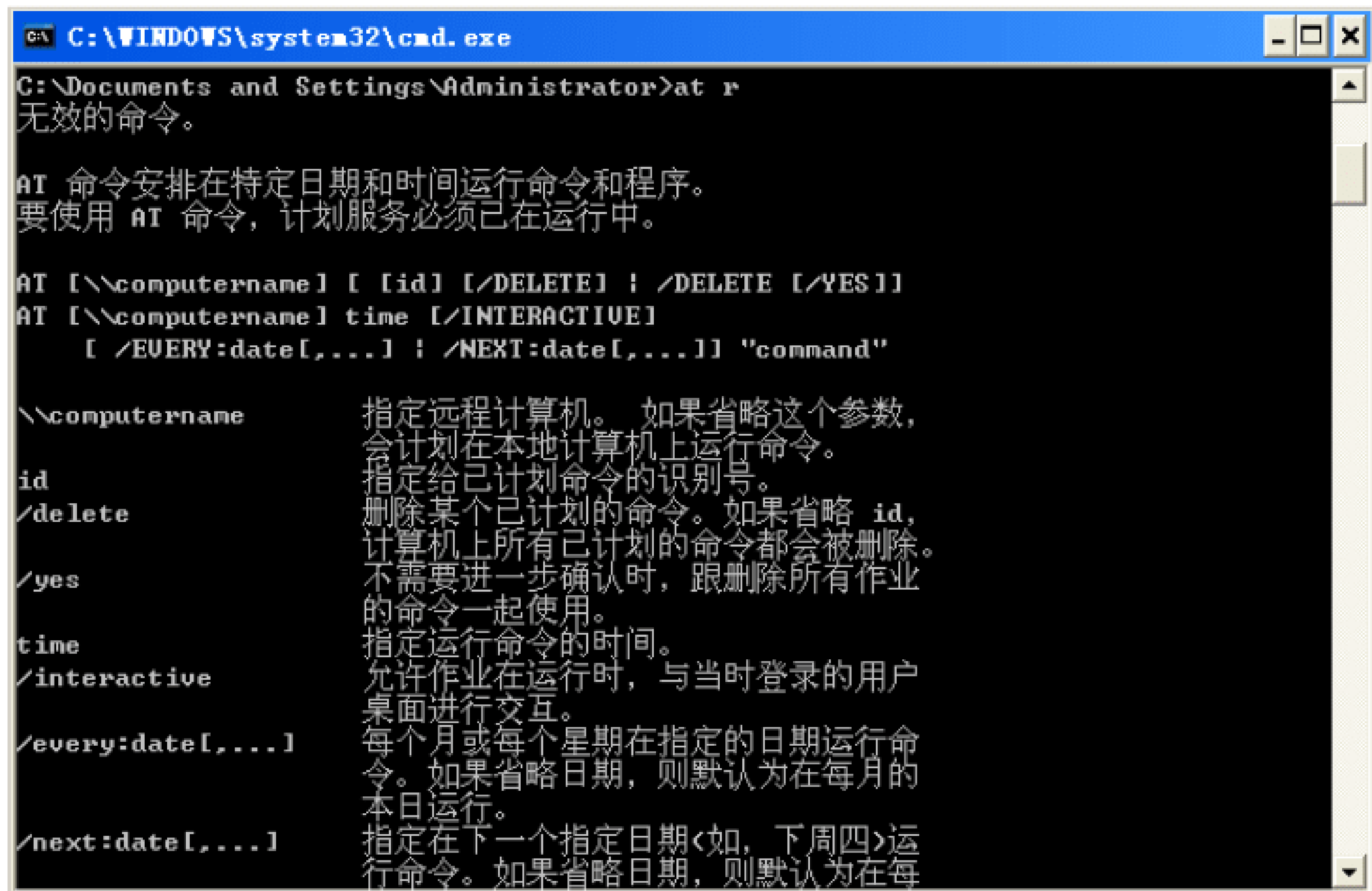


图 14.27　AT 指令的相关帮助信息

注意： AT 指令几乎被所有的计算机及其辅助硬件所支持，并且通过 AT 指令可以利用计算机向任何一种硬件发送相应的 AT 指令以实现相应的功能。

14.3.2　AT 指令详解

在前面一节中，向用户大致介绍了 AT 指令的定义、作用及其发送方式等。为了使用户加深对 AT 指令的理解以及使用，在本节中将以表格的方式向用户介绍常用的 AT 指令

及其功能含义，如表 14.1 所示。

表 14.1　AT指令及其功能含义

功　能	AT命令格式	详 细 说 明
厂家认证	AT+CGMI	获得厂家的标识
模式认证	AT+CGMM	查询支持频段
修订认证	AT+CGMR	查询软件版本
生产序号	AT+CGSN	查询IMEI NO
TE设置	AT+CSCS	选择支持网络
查询IMSI	AT+CIMI	查询国际移动电话支持认证
卡的认证	AT+CCID	查询SIM卡的序列号
功能列表	AT+GCAP	查询可供使用的功能列表
重复操作	A/	重复最后一次操作
关闭电源	AT+CPOF	暂停模块软件运行
设置状态	AT+CFUN	设置模块软件的状态
活动状态	AT+CPAS	查询模块当前活动状态
报告错误	AT+CMEE	报告模块设备错误
键盘控制	AT+CKPD	用字符模拟键盘操作
拨号命令	ATD	拨打电话号码
挂机命令	ATH	挂机
回应呼叫	ATA	当模块被呼叫时回应呼叫
详细错误	AT+CEER	查询错误的详细原因
DTMF信号	AT+VTD，+VTS	+VTD设置长度，+VTS发送信号
重复呼叫	ATDL	重复拨叫最后一次号码
自动拨号	AT%Dn	设备自动拨叫号码
自动接应	ATS0	模块自动接听呼叫
呼入载体	AT+CICB	查询呼入的模式，DATA or FAX or SPEECH
增益控制	AT+VGR，+VGT	+VGR调整听筒增益，+VGT调整话筒增益
静音控制	AT+CMUT	设置话筒静音
声道选择	AT+SPEAKER	选择不同声道（2对听筒和话筒）
回声取消	AT+ECHO	根据场所选择不同回声程度
单音修改	AT+SIDET	选择不同回声程度
初始声音参数	AT+VIP	恢复到厂家对声音参数的默认设置
信号质量	AT+CSQ	查询信号质量
网络选择	AT+COPS	设置选择网络方式（自动/手动）
网络注册	AT+CREG	当前网络注册情况
网络名称	AT+WOPN	查询当前使用网络提供者
网络列表	AT+CPOL	查询可供使用的网络
输入PIN	AT+CPIN	输入PIN码
输入PIN2	AT+CPIN2	输入第二个PIN码
保存尝试	AT+CPINC	显示可能的各个PIN码

续表

功　　能	AT命令格式	详 细 说 明
简单上锁	AT+CLCK	用户可以锁住状态
改变密码	AT+CPWD	改变各个PIN码
选择电话簿	AT+CPBS	选择不同的记忆体上存储的电话簿
读取电话簿	AT+CPBR	读取电话簿目录
查找电话簿	AT+CPBF	查找所需电话目录
写入电话簿	AT+CPBW	增加电话簿条目
电话号码查找	AT+CPBP	查找所需电话号码
动态查找	AT+CPBN	查找电话号码的一种方式
用户号码	AT+CNUM	选择不同的本机号码（因网络服务支持不同）
避免电话簿初始化	AT+WAIP	选择是否防止电话簿初始化
选择短消息服务	AT+CSMS	选择是否打开短消息服务以及广播服务
短消息存储	AT+CPMS	选择短消息优先存储区域
短消息格式	AT+CMGF	选择短消息支持格式（TEXT or PDU）
保存设置	AT+CSAS	保存+CSCA and +CSMP参数设置
恢复设置	AT+CRES	恢复+CSCA and +CSMP参数设置
显示TEXT参数	AT+CSDH	显示当前TEXT模式下结果代码
新消息提示	AT+CNMI	选择当有新的短消息来时系统提示方式
读短消息	AT+CMGR	读取短消息
列短消息	AT+CMGL	将存储的短消息列表
发送短消息	AT+CMGS	发送短消息
写短消息	AT+CMGW	写短消息并保存在存储器中
从内存中发短消息	AT+CMSS	发送在存储器中保存的短消息
设置TEXT参数	AT+CSMP	设置在TEXT模式下条件参数
删除短消息	AT+CMGD	删除保存的短消息
服务中心地址	AT+CSCA	提供短消息服务中心的号码
选择广播类型	AT+CSCB	选择系统广播短消息的类型
广播标识符	AT+WCBM	读取SIM卡中系统广播标识符
短消息位置修改	AT+WMSC	修改短消息位置
短消息覆盖	AT+WMGO	写一条短消息放在第一个空位
呼叫转移	AT+CCFC	设置呼叫转移
呼入载体	AT+CLCK	锁定呼入载体以及限制呼入或呼出
修改SS密码	AT+CPWD	修改提供服务密码
呼叫等待	AT+CCWA	控制呼叫等待服务
呼叫线路限定	AT+CLIR	控制呼叫线路认证
呼叫线路显示	AT+CLIP	显示当前呼叫线路认证
已连接线路认证	AT+COLP	显示当前已连接线路认证
计费显示	AT+CAOC	报告当前费用
累计呼叫	AT+CACM	累计呼叫费用
累计最大值	AT+CAMM	设置累计最大值

续表

功　　能	AT命令格式	详 细 说 明
单位计费	AT+CPUC	设置单位费用以及通话计时
多方通话	AT+CHLD	保持或挂断某一通话线路（支持多方通话）
当前呼叫	AT+CLCC	列出当前呼叫
补充服务	AT+CSSN	设置呼叫增值服务
非正式补充服务	AT+CUSD	非正式的增值服务
保密用户	AT+CCUG	选择是否在保密状态
载体选择	AT+CBST	选择数据传输的类型
选择模式	AT+FCLASS	选择发送数据or 传真
服务报告控制	AT+CR	是否报告提供服务
结果代码	AT+CRC	报告不同的结果代码(传输方式、语音或数据)
设备速率报告	AT+ILRR	是否报告当前传输速率
协议参数	AT+CRLP	设置无线连接协议参数
其他参数	AT+DOPT	设置其他的无线连接协议参数
传输速度	AT+FTM	设置传真发送的速度
接收速度	AT+FRM	设置传真接收的速度
HDLC传输速度	AT+FTH	设置传真发送的速度（使用HDLC协议）
HDLC接收速度	AT+FRH	设置传真接收的速度（使用HDLC协议）
停止传输并等待	AT+FTS	停止传真的发送并等待
静音接收	AT+FRS	保持一段静音等待
固定终端速率	AT+IPR	设置数据终端设备速率
其他位符	AT+ICF	设置停止位、奇偶校验位
流量控制	AT+IFC	设置本地数据流量
设置DCD信号	AT&C	控制数据载体探测信号
设置DTR信号	AT&D	控制数据终端设备准备信号
设置DSR信号	AT&S	控制数据设备准备信号
返回在线模式	ATO	返回到数据在线模式
结果代码抑制	ATQ	是否模块回复结果代码
DCE回应格式	ATV	决定数据通信设备回应格式
默认设置	ATZ	恢复到默认设置
保存设置	AT&W	保存所有对模块的软件修改
自动测试	AT&T	自动测试软件
回应	ATE	是否可见输入字符
恢复厂家设置	AT&F	软件恢复到厂家设置
显示设置	AT&V	显示当前的一些参数的设置
认证信息	ATI	显示多种模块认证信息
区域环境描述	AT+CCED	用户获取区域参数
自动接收电平显示	AT+CCED	扩展到显示接收信号强度
一般显示	AT+WIND	无
在ME和MSC之间数据计算模式	AT+ALEA	无
数据计算模式	AT+CRYPT	无
键盘管理	AT+EXPKEY	无

续表

功　能	AT命令格式	详 细 说 明
PLMN上的信息	AT+CPLMN	无
模拟数字转换测量	AT+ADC	无
模块事件报告	AT+CMER	无
选择语言	AT+WLPR	选择可支持的语言
增加语言	AT+WLPW	增加可支持的语言
读GPIO值	AT+WIOR	无
写GPIO值	AT+WIOW	无
放弃命令	AT+WAC	用于放弃SMS、SS and PLMN
设置单音	AT+WTONE	设置音频信号（WMOi3）
设置DTMF音	AT+WDTMF	设置DTMF音（WMOi3）

本节主要向用户讲解了一些常用的 AT 指令的基本格式以及这些 AT 指令的功能含义。通过本节的学习，用户可以知道常用短信猫的直接指令操作方法等。

一般情况下，用户可以方便地使用这些 AT 指令直接操作计算机辅助硬件设备等。例如，用户向短信猫发送 AT 指令，以获取 SIM 卡的序列号，其具体操作指令如下：

```
AT+CCID                    //获取 SIM 卡的序列号
```

当短信猫接收到该指令以后，会将 SIM 卡的序列号返回。这样，用户程序便可以从串口等数据缓冲区中读取这一数据并显示即可。

注意：用户实际使用 AT 指令时，需要结合实际硬件生产商的相关说明文档进行指令的格式化。这是因为不同的硬件生产商可能会有不同的 AT 指令格式。但是，这些 AT 指令格式大体上是一样的。

14.4　封装数据结构

前面的内容已经向用户讲解了关于短信猫与 PC 配合发送短消息的相关硬件设施以及 AT 指令等内容。本节将向用户介绍在实例程序中常用的短消息数据结构的封装方法以及短消息类的封装方法等。

14.4.1　封装消息数据结构

首先，用户需要定义短消息的数据结构并且添加相应的关键成员变量，以方便短消息的发送等操作。因此，在本章实例中，定义了短消息数据结构并将其命名为 SM_PARAM。该数据结构定义如下：

```
typedef struct {
 char SCA[16];           //短消息服务中心号码（SMSC 地址）
 char TPA[16];           //目标号码或回复号码（TP-DA 或 TP-RA）
 char TP_PID;            //用户信息协议标识（TP-PID）
```

```
 char TP_DCS;              //用户信息编码方式（TP-DCS）
 char TP_SCTS[16];         //服务时间戳字符串（TP_SCTS），接收时用到
 char TP_UD[160];          //原始用户信息（编码前或解码后的 TP-UD）
 short index;              //短消息序号，在读取时用到
} SM_PARAM;
```

在该结构体中，包括了发送短消息所必须的一些信息。例如，短消息服务中心号码、短消息接收方号码以及短消息操作的时间等。

然后，用户可以在实例程序中，使用该数据结构进行短消息数据的初始化工作。代码如下：

```
...                                               //省略部分代码
SM_PARAM m_sm;                                    //定义结构体对象
m_sm. SCA="13800009564";                          //初始化短消息服务中心号码
m_sm. TPA="13547645285";                          //初始化短消息接收方号码
...                                               //省略部分代码
```

在代码中，用户首先定义了短消息数据结构 SM_PARAM 的对象 m_sm。然后，再为该结构体中各个成员变量进行初始化。

如果用户将该结构体中的各个数据变量初始化完成以后，便可以将该数据结构通过串口发送到短信猫中执行。代码如下：

```
...                                               //省略部分代码
WriteFile(com,&m_sm,sizeof(m_sm),0,NULL);         //将结构体数据写入串口
...                                               //省略部分代码
```

用户将数据写入相应的串口后，短信猫中的相应程序会定时检测串口缓冲区中是否存在数据。如果短信猫程序检测到串口缓冲区中数据后，便会读取这些数据并执行。

14.4.2　封装接收消息数据结构

当短信猫接收到对方发送或者回复的短消息时，会将该信息内容发送到计算机的串口缓冲区中，以便计算机读取该内容并显示。所以，用户需要定义一个相应的结构体，获取该信息内容。在本章实例中，将定义结构体 SM_BUFF 实现该功能，其定义如下：

```
typedef struct {
    int len;                                      //获取到的信息长度
    char data[16384];                             //获取到的信息内容
} SM_BUFF;
```

该结构体主要被用于接收短信猫发送的短消息内容或者短信猫的应答状态等。例如，用户封装一个函数 gsmGetResponse()。在该函数中，用户需要判断短信猫的应答状态。代码如下：

```
int gsmGetResponse(SM_BUFF* pBuff)            //判断短信猫的应答状态
{
 int nLength;                                 //串口收到的数据长度
 int nState;                                  //定义应答状态变量
 nLength=ReadComm(&pBuff->data[pBuff->len], 128);
                                              //从串口读数据，追加到缓冲区尾部
```

```
 pBuff->len += nLength;                          //自动扩展信息长度
 nState=GSM_WAIT;                                //确定短信猫的应答状态
 if ((nLength > 0) && (pBuff->len >= 4))         //判断读取到的串口数据不为空
{
     if (strncmp(&pBuff->data[pBuff->len - 4], "OK\r\n", 4) == 0)
                                                 //比较两个字符串
            {
                nState=GSM_OK;                   //如果相同,则返回应答状态为 GSM_OK
                }
else                                             //如果不相同,则返回应答状态为 GSM_
                                                   ERR
      {
if (strstr(pBuff->data, "+CMS ERROR") != NULL)
       {
             nState=GSM_ERR;                     //设置短信猫的应答状态为 GSM_ERR
         }
     }
}
return nState;                                   //返回短信猫的应答状态
}
```

在代码中，用户主要是将结构体 SM_BUFF 中的成员变量 len 与串口读取的数据长度相加，得到最新的数据长度。然后，再判断获取到的数据中是否存在需要处理的关键字。例如，"OK\r\n"等。

注意：用户在使用该结构体时，需要首先在实例程序中定义一个全局的结构体变量。否则，用户对该结构体的所有调用都会失败。

14.5　封装短消息类

在前面的两个小节中，向用户讲述了相应数据结构体的封装方式以及各个数据变量的含义，并且使用实例说明了相应结构体的使用方法。在本节中，将向用户讲解短消息类的封装方法等。

14.5.1　定义短消息操作函数和数据结构

在本节中，用户需要将短消息数据相关的数据结构或者操作函数封装到一个文件中，便于用户查找以及提高程序的运行效率等。本节中将该封装文件名设置为 Sms，其定义文件即头文件 Sms.h 定义如下：

```
...                              //用户信息编码方式
#define GSM_7BIT        0
#define GSM_8BIT        4
#define GSM_UCS2        8
                                 //应答状态
#define GSM_WAIT        0        //等待，不确定
#define GSM_OK          1        //OK
#define GSM_ERR        -1        //ERROR
typedef struct {                 //短消息参数结构。其中，字符串以\0 结尾
```

```
 char SCA[16];                    //短消息服务中心号码（SMSC 地址）
 char TPA[16];                    //目标号码或回复号码（TP-DA 或 TP-RA）
 char TP_PID;                     //用户信息协议标识（TP-PID）
 char TP_DCS;                     //用户信息编码方式（TP-DCS）
 char TP_SCTS[16];                //服务时间戳字符串（TP_SCTS），接收时用到
 char TP_UD[160];                 //原始用户信息（编码前或解码后的 TP-UD）
 short index;                     //短消息序号，在读取时用到
} SM_PARAM;
typedef struct {                  //读取应答的缓冲区
 int len;                         //数据长度
 char data[16384];                //数据内容
} SM_BUFF;
 int gsmBytes2String(const unsigned char* pSrc, char* pDst, int nSrcLength);
                                  //短消息相关操作函数
int gsmString2Bytes(const char* pSrc, unsigned char* pDst, int nSrcLength);
int gsmEncode7bit(const char* pSrc, unsigned char* pDst, int nSrcLength);
int gsmDecode7bit(const unsigned char* pSrc, char* pDst, int nSrcLength);
int gsmEncode8bit(const char* pSrc, unsigned char* pDst, int nSrcLength);
int gsmDecode8bit(const unsigned char* pSrc, char* pDst, int nSrcLength);
int gsmEncodeUcs2(const char* pSrc, unsigned char* pDst, int nSrcLength);
int gsmDecodeUcs2(const unsigned char* pSrc, char* pDst, int nSrcLength);
int gsmInvertNumbers(const char* pSrc, char* pDst, int nSrcLength);
int gsmSerializeNumbers(const char* pSrc, char* pDst, int nSrcLength);
int gsmEncodePdu(const SM_PARAM* pSrc, char* pDst);
int gsmDecodePdu(const char* pSrc, SM_PARAM* pDst);
BOOL gsmInit();                      //初始化短消息操作
Int gsmSendMessage(SM_PARAM* pSrc);
                                     //短消息发送
int gsmReadMessageList();            //短消息列表设置
int gsmDeleteMessage(int index);     //删除短消息
int gsmGetResponse(SM_BUFF* pBuff);  //获取短信猫的应答状态
int gsmParseMessageList(SM_PARAM* pMsg, SM_BUFF* pBuff);
                                     //从短消息列表中解析短消息
```

接下来，用户还需要在实现文件“Sms.cpp”中实现以上函数的相关功能并添加相应的功能代码。代码如下：

```
#include "stdafx.h"                               //包含相关的头文件
#include "Sms.h"
#include "Comm.h"
/*  可打印字符串转换为字节数据
如："C8329BFD0E01" --> {0xC8, 0x32, 0x9B, 0xFD, 0x0E, 0x01}
输入：pSrc - 源字符串指针
nSrcLength - 源字符串长度
输出：pDst - 目标数据指针
返回：目标数据长度*/
int gsmString2Bytes(const char* pSrc, unsigned char* pDst, int nSrcLength)
{
 for (int i=0; i < nSrcLength; i += 2)
{
      if ((*pSrc >= '0') && (*pSrc <= '9'))     //输出源字符串的高 4 位
      {
             *pDst=(*pSrc - '0') << 4;          //将源字符串向左移动 4 位
```

```
        }
        else
        {
              *pDst=(*pSrc - 'A' + 10) << 4;
         }
         pSrc++;                              //源字符串指针自加
         if ((*pSrc>='0') && (*pSrc<='9'))    //输出源字符串的低 4 位
         {
              *pDst |= *pSrc - '0';
        }
        else
        {
             *pDst |= *pSrc - 'A' + 10;
        }
        pSrc++;
        TRACE("%s",pSrc);                     //向调试器输出窗口输出源字符串
        pDst++;
        TRACE("%s",pDst);                     //向调试器输出窗口输出目标字符串
}
 return (nSrcLength / 2);                     //返回目标数据长度
}
/*  字节数据转换为可打印字符串
如: {0xC8, 0x32, 0x9B, 0xFD, 0x0E, 0x01} --> "C8329BFD0E01"
输入: pSrc - 源数据指针
nSrcLength - 源数据长度
输出: pDst - 目标字符串指针
返回: 目标字符串长度      */
int gsmBytes2String(const unsigned char* pSrc, char* pDst, int nSrcLength)
{
 const char tab[]="0123456789ABCDEF";        //定义 0x0～0xf 字符查找表
 for (int i=0; i < nSrcLength; i++)          //遍历源字符串
{
      *pDst++=tab[*pSrc >> 4];               //输出源字符串的高 4 位
      *pDst++=tab[*pSrc & 0x0f];             //输出源字符串的低 4 位
     pSrc++;
}
*pDst='\0';                                   //为输出字符串添加结束符\0
 return (nSrcLength * 2);                     //返回目标字符串长度
}
/*  7bit 编码                                  //函数说明
输入: pSrc - 源字符串指针
nSrcLength - 源字符串长度
输出: pDst - 目标编码串指针
返回: 目标编码串长度      */
int gsmEncode7bit(const char* pSrc, unsigned char* pDst, int nSrcLength)
{
 int nSrc;                                    //源字符串的计数值
 int nDst;                                    //目标编码串的计数值
 int nChar;                  //当前正在处理的组内字符字节的序号，范围是 0～7
 unsigned char nLeft;                         //上一字节残余的数据
 nSrc=0;                                      //计数值初始化
 nDst=0;
 while (nSrc < nSrcLength)
 {
        nChar=nSrc & 7;                       //取源字符串的计数值的最低 3 位
        if(nChar == 0)                        //处理源串的每个字节
```

```
      {                         //组内第一个字节，只是保存起来，待处理下一个字节时使用
            nLeft=*pSrc;
      }
      else
     {                              //组内其他字节，将其右边部分与残余数据相加，得到
                                      一个目标编码字节
           *pDst=(*pSrc << (8-nChar)) | nLeft;
                                    //将该字节剩下的左边部分，作为残余数据保存起来
           nLeft=*pSrc >> nChar;
                                    //修改目标串的指针和计数值
           pDst++;                  //自加
           nDst++;
        }
        pSrc++;                     //修改源串的指针和计数值
        nSrc++;
}
return nDst;                        //返回目标串长度
}
/*  7bit 解码
输入: pSrc - 源编码串指针
nSrcLength - 源编码串长度
输出: pDst - 目标字符串指针
返回: 目标字符串长度        */
int gsmDecode7bit(const unsigned char* pSrc, char* pDst, int nSrcLength)
{
 int nSrc;                          //源字符串的计数值
 int nDst;                          //目标解码串的计数值
 int nByte;                         //当前正在处理的组内字节的序号，范围是 0～6
 unsigned char nLeft;               //上一字节残余的数据
 nSrc=0;                            //计数值初始化
 nDst=0;
 nByte=0;                           //组内字节序号和残余数据初始化
 nLeft=0;
                                    //将源数据每 7 个字节分为一组，解压缩成 8 个字节
                                    //循环该处理过程，直至源数据被处理完
                                    //如果分组不到 7 字节，也能正确处理
 while(nSrc<nSrcLength)
{
                                    //将源字节右边部分与残余数据相加，去掉最高位，
                                       得到一个目标解码字节
      *pDst=((*pSrc << nByte) | nLeft) & 0x7f;
                                    //将该字节剩下的左边部分，作为残余数据保存起来
        nLeft=*pSrc >> (7-nByte);
        pDst++;                     //修改目标串的指针和计数值
        nDst++;
        nByte++;                    //修改字节计数值
        if(nByte == 7)              //如果循环到了一组的最后一个字节
        {
            *pDst=nLeft;            //额外得到一个目标解码字节
            pDst++;                 //修改目标串的指针和计数值
            nDst++;
            nByte=0;                //组内字节序号和残余数据初始化
            nLeft=0;
        }
        pSrc++;                     //修改源串的指针和计数值
        nSrc++;
```

```
}
*pDst='\0';                                          //为输出字符串添加结束符\0
return nDst;                                         //返回目标串长度
}
/*  8bit 编码
输入: pSrc - 源字符串指针
nSrcLength - 源字符串长度
输出: pDst - 目标编码串指针
返回: 目标编码串长度        */
int gsmEncode8bit(const char* pSrc, unsigned char* pDst, int nSrcLength)
{
 memcpy(pDst, pSrc, nSrcLength);                     //复制字符串
 return nSrcLength;                                  //返回处理字符串的长度值
}
/*  8bit 解码
输入: pSrc - 源编码串指针
nSrcLength -  源编码串长度
输出: pDst -  目标字符串指针
返回: 目标字符串长度        */
int gsmDecode8bit(const unsigned char* pSrc, char* pDst, int nSrcLength)
{
 memcpy(pDst, pSrc, nSrcLength);                     //复制字符串
 *pDst='\0';                                         //为输出字符串添加结束符
 return nSrcLength;                                  //返回处理字符串的长度值
}
/*  UCS2 编码
输入: pSrc - 源字符串指针
nSrcLength - 源字符串长度
输出: pDst - 目标编码串指针
返回: 目标编码串长度        */
int gsmEncodeUcs2(const char* pSrc, unsigned char* pDst, int nSrcLength)
{
 int nDstLength;                                     //定义 UNICODE 宽字符数目
 WCHAR wchar[128];                                   //定义 UNICODE 串缓冲区
 nDstLength=MultiByteToWideChar(CP_ACP, 0, pSrc, nSrcLength, wchar, 128);
                                                     //将字符串转化为 UNICODE 串
 for(int i=0; i<nDstLength; i++)                     //高低字节对调并输出
 {
        *pDst++=wchar[i] >> 8;                       //先输出高位字节
        *pDst++=wchar[i] & 0xff;                     //后输出低位字节
}
TRACE("%s",wchar);                                   //判断处理是否成功
 return nDstLength * 2;                              //返回目标编码串长度
}
/*  UCS2 解码
输入: pSrc - 源编码串指针
nSrcLength -  源编码串长度
输出: pDst -  目标字符串指针
返回: 目标字符串长度        */
int gsmDecodeUcs2(const unsigned char* pSrc, char* pDst, int nSrcLength)
{
 int nDstLength;                                     //定义 UNICODE 宽字符数目
 WCHAR wchar[128];                                   //定义 UNICODE 串缓冲区
 for(int i=0; i<nSrcLength/2; i++)                   //高低字节对调，拼成 UNICODE
```

```
 {
       wchar[i]=*pSrc++ << 8;               //先输出高位字节
       wchar[i] |= *pSrc++;                 //后输出低位字节
}
 nDstLength=WideCharToMultiByte(CP_ACP, 0, wchar, nSrcLength/2, pDst, 160,
NULL, NULL);
                                             //将 UNICODE 字符串转换为字符串
 pDst[nDstLength]='\0';                      //为输出字符串添加结束符
 return nDstLength;                          //返回目标字符串的长度值
}
/*  正常顺序的字符串转换为两两颠倒的字符串，若长度为奇数，补 F 凑成偶数
如："8613851872468" --> "683158812764F8"
输入：pSrc - 源字符串指针
nSrcLength - 源字符串长度
输出：pDst - 目标字符串指针
返回：目标字符串长度        */
int gsmInvertNumbers(const char* pSrc, char* pDst, int nSrcLength)
{
 int nDstLength;                             //定义目标字符串长度
 char ch;                                    //定义字符变量，用于保存一个字符
 nDstLength=nSrcLength;                      //复制串长度值
 for(int i=0; i<nSrcLength;i+=2)             //改变字符串的排列顺序
 {
       ch=*pSrc++;                           //保存先出现的字符
       *pDst++=*pSrc++;                      //复制后出现的字符
       *pDst++=ch;                           //复制先出现的字符
}
if(nSrcLength && 1)                          //判断源串长度是否为奇数
{
      *(pDst-2)='F';                         //如果是奇数，那么补 F
      nDstLength++;                          //目标串长度自加 1
 }
 *pDst='\0';                                 //为输出字符串添加结束符
 return nDstLength;                          //返回目标字符串长度
}
/*  将顺序调整后的字符串转换为正常顺序

的字符串如："683158812764F8" -->
"8613851872468"
输入：pSrc - 源字符串指针
nSrcLength - 源字符串长度
输出：pDst - 目标字符串指针
返回：目标字符串长度        */
int gsmSerializeNumbers(const char* pSrc, char* pDst, int nSrcLength)
{
 int nDstLength;                             //定义目标字符串长度
 char ch;                                    //定义变量，用于保存一个字符
 nDstLength=nSrcLength;                      //复制串长度
 for(int i=0; i<nSrcLength;i+=2)             //改变字符串的顺序
 {
       ch=*pSrc++;                           //保存先出现的字符
       *pDst++=*pSrc++;                      //复制后出现的字符
       *pDst++=ch;                           //复制先出现的字符
}
```

```
if(*(pDst-1) == 'F')                    //判断字符串最后的字符是否为 F
{
       pDst--;                          //将字符串自减
       nDstLength--;                    //目标字符串长度减 1
 }
 *pDst='\0';                            //为输出字符串添加结束符
 return nDstLength;                     //返回目标字符串长度
}
/*  PDU 编码，用于编制、发送短消息
输入：pSrc - 源 PDU 参数指针
输出：pDst - 目标 PDU 串指针
返回：目标 PDU 串长度      */
int gsmEncodePdu(const SM_PARAM* pSrc, char* pDst)
{
 int nLength;                           //内部用的串长度
 int nDstLength;                        //目标 PDU 串长度
 unsigned char buf[256];                //内部用的缓冲区
                                        //SMSC 地址信息段
 nLength=strlen(pSrc->SCA);             //SMSC 地址字符串的长度
 buf[0]=(char)((nLength & 1) == 0 ? nLength : nLength + 1) / 2 + 1;
                                        //SMSC 地址信息长度
 buf[1]=0x91;                           //固定：用国际格式号码
 nDstLength=gsmBytes2String(buf, pDst, 2);
                                        //转换 2 个字节到目标 PDU 串
 nDstLength += gsmInvertNumbers(pSrc->SCA, &pDst[nDstLength], nLength);
                                        //转换 SMSC 号码到目标 PDU 串
                                        //TPDU 段基本参数、目标地址等
 nLength=strlen(pSrc->TPA);             //TP-DA 地址字符串的长度
 buf[0]=0x11;                           //发送短信（TP-MTI=01），TP-VP 用相对格式
                                          （TP-VPF=10）
 buf[1]=0;                              //TP-MR=0
 buf[2]=(char)nLength;                  //目标地址数字个数（TP-DA 地址字符串真实长
度）
 buf[3]=0x91;                           //固定：用国际格式号码
 nDstLength += gsmBytes2String(buf, &pDst[nDstLength], 4);
                                        //转换 4 个字节到目标 PDU 串
 nDstLength  += gsmInvertNumbers(pSrc->TPA, &pDst[nDstLength], nLength);

                                        //转换 TP-DA 到目标 PDU 串
                                        //TPDU 段协议标识、编码方式、用户信息等
 nLength=strlen(pSrc->TP_UD);           //用户信息字符串的长度
 buf[0]=pSrc->TP_PID;                   //协议标识（TP-PID）
 buf[1]=pSrc->TP_DCS;                   //用户信息编码方式（TP-DCS）
 buf[2]=0;                              //有效期（TP-VP）为 5 分钟
 if(pSrc->TP_DCS == GSM_7BIT)           //判断编码方式
 {
                                        //7-bit 编码方式
      buf[3]=nLength;                   //编码前的长度值
      nLength=gsmEncode7bit(pSrc->TP_UD, &buf[4], nLength+1) + 4;
                                        //转换 TP-DA 到目标 PDU 串
}
else
if(pSrc->TP_DCS == GSM_UCS2)            //UCS2 编码方式
{
```

```
      buf[3]=gsmEncodeUcs2(pSrc->TP_UD, &buf[4], nLength);
                                    //转换 TP-DA 到目标 PDU 串
      nLength=buf[3] + 4;           //nLength 等于该段数据长度
}
else
{
                                    //8-bit 编码方式
     buf[3]=gsmEncode8bit(pSrc->TP UD, &buf[4], nLength);
                                    //转换 TP-DA 到目标 PDU 串
     nLength=buf[3] + 4;            //nLength 等于该段数据长度
 }
 nDstLength += gsmBytes2String(buf, &pDst[nDstLength], nLength);
                                    //转换该段数据到目标 PDU 串
 return nDstLength;                 //返回目标字符串长度
}
/*  PDU 解码，用于接收、阅读短消息
输入: pSrc - 源 PDU 串指针
输出: pDst - 目标 PDU 参数指针
返回: 用户信息串长度        */
int gsmDecodePdu(const char* pSrc, SM_PARAM* pDst)
{
 int nDstLength;                    //定义目标 PDU 串长度
 unsigned char tmp;                 //定义内部用的临时字节变量
 unsigned char buf[256];            //定义内部用的缓冲
 gsmString2Bytes(pSrc, &tmp, 2);    //取 SMSC 地址信息段字符串长度值
 tmp=(tmp - 1) * 2;                 //SMSC 号码串长度
 pSrc += 4;                         //将指针后移，忽略了 SMSC 地址格式
 gsmSerializeNumbers(pSrc, pDst->SCA, tmp);
                                    //转换 SMSC 号码到目标 PDU 串
 pSrc += tmp;                       //将指针后移
                                    //TPDU 段基本参数
 gsmString2Bytes(pSrc, &tmp, 2);    //取基本参数
 pSrc += 2;                         //将指针后移
 gsmString2Bytes(pSrc, &tmp, 2);    //取出回复号码长度
 if(tmp & 1) tmp += 1;              //调整奇偶性
 pSrc += 4;                         //指针后移，忽略了回复地址（TP-RA）格式
 gsmSerializeNumbers(pSrc, pDst->TPA, tmp);
                                    //取 TP-RA 号码
 pSrc += tmp;                       //指针后移
                                    //TPDU 段协议标识、编码方式、用户信息等
 gsmString2Bytes(pSrc, (unsigned char*)&pDst->TP_PID, 2);
                                    //取协议标识（TP-PID）
 pSrc += 2;                         //将指针后移
 gsmString2Bytes(pSrc, (unsigned char*)&pDst->TP_DCS, 2);
                                    //取编码方式（TP-DCS）
 pSrc += 2;                         //指针后移
 gsmSerializeNumbers(pSrc, pDst->TP_SCTS, 14);
                                    //服务时间戳字符串（TP_SCTS）
 pSrc += 14;                        //将指针后移
 gsmString2Bytes(pSrc, &tmp, 2);    //用户信息长度（TP-UDL）
 pSrc += 2;                         //将指针后移
 if(pDst->TP_DCS == GSM_7BIT)       //7-bit 解码方式
 {
      nDstLength=gsmString2Bytes(pSrc, buf, tmp & 7 ? (int)tmp * 7 / 4 + 2 :
(int)tmp * 7 / 4);
```

```
                                              //进行格式转换
      gsmDecode7bit(buf, pDst->TP_UD, nDstLength);
                                              //转换到 TP-DU
      nDstLength=tmp;
 }
 else
 if(pDst->TP_DCS == GSM_UCS2)                 //UCS2 解码
{
       nDstLength=gsmString2Bytes(pSrc, buf, tmp * 2);
                                              //进行格式转换
       nDstLength=gsmDecodeUcs2(buf, pDst->TP_UD, nDstLength);
                                              //转换到 TP-DU
 }
 else
 {
                                              //8-bit 解码
     nDstLength=gsmString2Bytes(pSrc, buf, tmp * 2);
                                              //格式转换
     nDstLength=gsmDecode8bit(buf, pDst->TP_UD, nDstLength);
                                              //转换到 TP-DU
 }
 return nDstLength;                           //返回目标字符串长度
}
BOOL gsmInit()                                //初始化 GSM 状态
{
 char ans[128];                               //定义应答字符数组
                                              //测试 GSM-MODEM 的存在性
 WriteComm("AT\r", 3);                        //向串口写入 AT 指令字符串
 ReadComm(ans, 128);                          //读取串口缓冲区中的数据
 if (strstr(ans, "OK") == NULL)               //判断字符串
         return FALSE;
 WriteComm("ATE0\r", 5);                      // ECHO OFF
 ReadComm(ans, 128);
 WriteComm("AT+CMGF=0\r", 10);                //PDU 模式
 ReadComm(ans, 128);
 return TRUE;
}
/*  发送短消息，仅发送命令，不读取应答
输入: pSrc - 源 PDU 参数指针   */
int gsmSendMessage(SM_PARAM* pSrc)
{
 int nPduLength;                              //定义 PDU 串长度
 unsigned char nSmscLength;                   //定义变量，用于 SMSC 串长度
 int nLength;                                 //串口收到的数据长度
 char cmd[16];                                //定义命令串
 char pdu[512];                               //PDU 串
 char ans[128];                               //应答串
 nPduLength=gsmEncodePdu(pSrc, pdu);          //根据 PDU 参数，编码 PDU 串
 strcat(pdu, "\x01a");                        //以快捷键 Ctrl+Z 结束
 gsmString2Bytes(pdu, &nSmscLength, 2);       //取 PDU 串中的 SMSC 信息长度
 nSmscLength++;                               //加上长度字节本身
                                 //命令中的长度，不包括 SMSC 信息长度以数据字节计
 sprintf(cmd, "AT+CMGS=%d\r", nPduLength / 2 - nSmscLength); //生成命令
TRACE("%s", cmd);
 TRACE("%s\n", pdu);
 WriteComm(cmd, strlen(cmd));                 //将命令字符串写入串口
```

```
 nLength=ReadComm(ans, 128);                       //从串口读取应答数据
 if(nLength == 4 && strncmp(ans, "\r\n> ", 4) == 0)
                                                    //根据能否找到"\r\n>"决定成功与否
 {
      return WriteComm(pdu, strlen(pdu));           //得到肯定回答，继续输出 PDU 串
 }
 return 0;
}
/*  读取短消息，仅发送命令，不读取应答
 用+CMGL 代替+CMGR，可一次性读出
 全部短消息        */
 int gsmReadMessageList()
 {
  return WriteComm("AT+CMGL\r", 8);                //写入串口数据
  }
  /* 删除短消息，仅发送命令，不读取应答

  输入: index - 短消息序号，1-255      */
  int gsmDeleteMessage(int index)
  {
   char cmd[16];                                    //定义命令字符数组
   sprintf(cmd, "AT+CMGD=%d\r", index);             //生成命令
   return WriteComm(cmd, strlen(cmd));              //写入串口数据
 }
 /* 读取 GSM MODEM 的应答，可能是一部

分
 输出: pBuff - 接收应答缓冲区
 返回: GSM MODEM 的应答状态, GSM_WAIT/GSM_OK /GSM_ERR
 备注: 可能需要多次调用才能完成读取一次应
 答，首次调用时应将 pBuff 初始化      */
 int gsmGetResponse(SM_BUFF* pBuff)
 {
  int nLength;                                      //定义串口收到的数据长度
  int nState;
                                                    //从串口读数据，追加到缓冲区尾部
  nLength=ReadComm(&pBuff->data[pBuff->len], 128);
  pBuff->len += nLength;
  nState=GSM_WAIT;                                  //确定 GSM MODEM 的应答状态
  if ((nLength > 0) && (pBuff->len >= 4))
  {
        if (strncmp(&pBuff->data[pBuff->len - 4], "OK\r\n", 4) == 0)
               nState=GSM_OK;
        else
        if (strstr(pBuff->data, "+CMS ERROR") != NULL)
                     nState=GSM_ERR;
 }
 return nState;                                     //返回状态标志
}
/*  从列表中解析出全部短消息
输入: pBuff - 短消息列表缓冲区
输出: pMsg - 短消息缓冲区
返回: 短消息条数      */
int gsmParseMessageList(SM_PARAM* pMsg, SM_BUFF* pBuff)
{
 int nMsg;                                          //定义短消息计数值
```

```
 char* ptr;                              //定义内部用的数据指针
 nMsg=0;
 ptr=pBuff->data;
 while((ptr=strstr(ptr, "+CMGL:")) != NULL)
                                     //循环读取每一条短消息, 以“+CMGL”:开头
 {
      ptr += 6;                          //跳过“+CMGL”:, 定位到序号
      sscanf(ptr, "%d", &pMsg->index); //读取序号
      ptr=strstr(ptr, "\r\n");           //查找下一行
      if (ptr != NULL)
      {
           ptr += 2;                     //跳过"\r\n", 定位到 PDU
           gsmDecodePdu(ptr, pMsg);      //PDU 串解码
           pMsg++;                       //准备读下一条短消息
           nMsg++;                       //短消息计数加 1
        }
 }
 return nMsg;                            //返回短消息计数
}
```

在本节中，用户主要是在实例程序中，将所有的关于短消息的操作函数或结构体全部封装在统一的文件中。这样，用户使用时比较方便。例如，在上面的程序中，函数 ReadComm()是专门用于操作串口的函数。因此，在 14.5.2 节中，将向用户介绍怎样将串口操作的相关函数进行封装。

14.5.2　定义串口操作函数

首先，用户需要在实例程序中，创建一个新文件，名称修改为 Comm.h。该文件的作用是声明相关的串口操作函数。其具体代码如下：

```
#if !defined(COMM H  )
#define COMM H
BOOL OpenComm(const char* pPort, int nBaudRate=57600, int nParity=NOPARITY,
int nByteSize=8, int nStopBits=ONESTOPBIT);                //打开串口
BOOL CloseComm();                                          //关闭串口
int ReadComm(void* pData, int nLength);                    //读取串口
int WriteComm(void* pData, int nLength);                   //写入串口
#endif
```

在头文件 Comm.h 中，用户声明了常用的串口操作函数的原型。

然后，用户在实例程序中创建一个文件，名称修改为 Comm.cpp。该文件的作用是实现串口操作相关函数的定义。代码如下：

```
#include "stdafx.h"                //包含头文件
#include "Comm.h"
HANDLE hComm;                      //定义串口设备句柄

/*  打开串口
输入: pPort - 串口名称或设备路径, 可
用"COM1"或"\\.\COM1"两种方式, 本书
建议使用后者
nBaudRate - 波特率
```

```
nParity - 奇偶校验
nByteSize - 数据字节宽度
nStopBits - 停止位        */
BOOL OpenComm(const char* pPort, int nBaudRate, int nParity, int nByteSize,
int nStopBits)
{
 DCB dcb;                        //串口控制块
 COMMTIMEOUTS timeouts={         //串口超时控制参数
    100,                         //读字符间隔超时时间: 100 ms
    1,                           //读操作时每字符的时间: 1 ms (n个字符总共为n ms)
    500,                         //基本的(额外的)读超时时间: 500 ms
    1,                           //写操作时每字符的时间: 1 ms (n个字符总共为n ms)
    100};                        //基本的(额外的)写超时时间: 100 ms
 hComm=CreateFile(pPort,         //串口名称或设备路径
        GENERIC_READ | GENERIC_WRITE,
                                 //读写方式
        0,                       //设置共享方式为独占
        NULL,                    //默认的安全描述符
        OPEN_EXISTING,           //创建方式
        0,                       //使用默认文件属性
        NULL);                   //不需要模板文件

 if(hComm == INVALID HANDLE VALUE) return FALSE;
                                 //打开串口失败
 GetCommState(hComm, &dcb);      //获取串口信息并存入 DCB 数据结构中
 dcb.BaudRate=nBaudRate;         //为 DCB 结构体重新赋值
 dcb.ByteSize=nByteSize;
 dcb.Parity=nParity;
 dcb.StopBits=nStopBits;
 SetCommState(hComm,&dcb);       //将串口信息存入 DCB 结构体中
 SetupComm(hComm,4096,1024);     //设置输入输出缓冲区大小
 SetCommTimeouts(hComm,&timeouts); //设置超时
 return TRUE;                    //返回 TRUE
}
BOOL CloseComm()                 //关闭串口的函数
{
 return CloseHandle(hComm);      //调用 API 函数关闭串口句柄
}
/*  写串口
输入: pData - 待写的数据缓冲区指针
nLength - 待写的数据长度
返回: 实际写入的数据长度        */
int WriteComm(void* pData, int nLength)
{
 DWORD dwNumWrite;               //串口发出的数据长度
 WriteFile(hComm, pData, (DWORD)nLength, &dwNumWrite, NULL);
 return (int)dwNumWrite;         //返回实际写入串口的字节数
}
/*  读串口
输入: pData - 待读的数据缓冲区指针
nLength - 待读的最大数据长度
返回: 实际读出的数据长度        */
int ReadComm(void* pData, int nLength)
{
```

```
 DWORD dwNumRead;                //串口收到的数据长度
 ReadFile(hComm, pData, (DWORD)nLength, &dwNumRead, NULL);
 return (int)dwNumRead;          //返回实际读取串口的字节数
}
```

用户使用以上代码时，只需要将头文件 Comm.h 包含到实例程序中即可。例如，包含本节中的头文件，代码如下：

```
#include "Comm.h"                            //包含头文件 Comm.h
```

用户使用上面的代码后，便可以在程序中调用文件 Comm.cpp 中的任何相关函数了。

注意：用户使用代码“#include”包含指定头文件时，如果该头文件的名称使用符号“《》”括起来，则表示该头文件为系统预定义的头文件。否则，表示用户自定义的头文件。

14.5.3　封装短消息类

在前面两节中，已经向用户详细介绍了串口操作相关函数和短消息操作相关函数的声明以及定义文件的封装。在本节中，将使用前面封装的相关文件和功能函数实现短消息类的定义和功能实现等。

1．定义短消息类

首先，用户需要在实例工程中，对短消息类进行定义并修改该类名为 CSmsTraffic。其具体定义代码如下：

```
#if  !defined(AFX SMSTRAFFIC H  3A4D81DE C363 42D6 8A47 3BA017BFBF56  IN
CLUDED_)
#define
AFX SMSTRAFFIC H  3A4D81DE C363 42D6 8A47 3BA017BFBF56  INCLUDED
#if _MSC_VER > 1000
#pragma once
#endif //  MSC VER > 1000
#include "Sms.h"                            //包含所需头文件
#include "Comm.h"
#define MAX_SM_SEND    128                  //发送队列长度
#define MAX SM RECV    128                  //接收队列长度
class CSmsTraffic                           //定义短消息类
{
public:
 CSmsTraffic();                             //构造函数
 virtual ~CSmsTraffic();
 int m_nSendIn;                             //发送队列的输入指针
 int m_nSendOut;                            //发送队列的输出指针
 int m_nRecvIn;                             //接收队列的输入指针
 int m_nRecvOut;                            //接收队列的输出指针
 SM_PARAM m_SmSend[MAX_SM_SEND];            //发送短消息队列
 SM_PARAM m_SmRecv[MAX_SM_SEND];            //接收短消息队列
 CRITICAL_SECTION m_csSend;                 //与发送相关的临界段
 CRITICAL_SECTION m_csRecv;                 //与接收相关的临界段
 HANDLE m_hKillThreadEvent;                 //通知子线程关闭的事件
```

```
 HANDLE m_hThreadKilledEvent;              //子线程关闭时产生的事件
 void PutSendMessage(SM_PARAM* pSmParam);
                                           //将短消息放入发送队列
 BOOL GetSendMessage(SM_PARAM* pSmParam);
                                           //从发送队列中取一条短消息
 void PutRecvMessage(SM_PARAM* pSmParam, int nCount);
                                           //将短消息放入接收队列
 BOOL GetRecvMessage(SM_PARAM* pSmParam);
                                           //从接收队列中取一条短消息
 static UINT SmThread(LPVOID lpParam);     //短消息收发处理子线程
};
#endif                                     //短消息类定义完毕
```

用户通过上面的代码可以清楚地了解到关于短消息类 CSmsTraffic 的常用函数的定义方式。这样，用户使用该类的成员函数时，可以很快知道所调用的函数相关类型等。

注意：用户在定义短消息类时，一定要在类定义前，在程序中添加代码包含该类所需要的头文件。

2．实现短消息类

现在，用户可以在程序中实现短消息类中的所有功能函数的基本功能了。用户在实例工程中，新建一个文件并命名为 SmsTraffic.cpp，然后便可以在该文件中添加实现函数功能的代码了。短消息类的成员函数实现代码如下：

```
#include "stdafx.h"                          //包含相关头文件
#include "SmsTest.h"
#include "SmsTraffic.h"
#ifdef _DEBUG
#undef THIS_FILE
static char THIS_FILE[]=__FILE__;
#define new DEBUG_NEW
#endif
///////////////////////////////////////////////////////////////////////
短消息类成员函数的基本实现
///////////////////////////////////////////////////////////////////////
CSmsTraffic::CSmsTraffic()                   //短消息类构造函数实现
{
 m_nSendIn=0;                                //初始化该类中的数据成员
 m_nSendOut=0;
 m_nRecvIn=0;
 m_nRecvOut=0;
 m_hKillThreadEvent=CreateEvent(NULL, TRUE, FALSE, NULL);
                                             //创建事件对象
 m_hThreadKilledEvent=CreateEvent(NULL, TRUE, FALSE, NULL);
 InitializeCriticalSection(&m_csSend);       //初始化临界区对象
 InitializeCriticalSection(&m_csRecv);

 AfxBeginThread(SmThread, this, THREAD_PRIORITY_NORMAL);
                                             //启动子线程
}
CSmsTraffic::~CSmsTraffic()                  //短消息类的析构函数实现
{
 SetEvent(m_hKillThreadEvent);               //设置事件对象为有信号状态
```

```
 WaitForSingleObject(m_hThreadKilledEvent, INFINITE);
                                          //等待子线程关闭
 DeleteCriticalSection(&m_csSend);        //删除临界区对象
 DeleteCriticalSection(&m_csRecv);
 CloseHandle(m_hKillThreadEvent);         //关闭事件句柄
 CloseHandle(m_hThreadKilledEvent);
}
void CSmsTraffic::PutSendMessage(SM_PARAM* pparam)
                                          //放入一条短消息到发送队列
{
 EnterCriticalSection(&m_csSend);         //进入临界区对象中
 memcpy(&m_SmSend[m_nSendIn], pparam, sizeof(SM_PARAM));
                                          //复制数据
 m_nSendIn++;                             //自加
 if (m_nSendIn >= MAX_SM_SEND)            //如果发送队列的长度大于规定值
       {
             m_nSendIn=0;                 //则将队列长度变量设置为 0
             }
 LeaveCriticalSection(&m_csSend);         //离开临界区
}
BOOL CSmsTraffic::GetSendMessage(SM_PARAM* pparam)
                                          //从发送队列中，取出一条短消息
{
 BOOL fSuccess=FALSE;                     //定义布尔变量并初始化
 EnterCriticalSection(&m_csSend);         //进入临界区对象
 if (m_nSendOut != m_nSendIn)             //判断队列长度是否相等
{
       memcpy(pparam, &m_SmSend[m_nSendOut], sizeof(SM_PARAM));
                                          //复制数据
       m_nSendOut++;                      //增加队列长度
       if (m_nSendOut >= MAX_SM_SEND)     //如果队列长度大于规定值
             {
                   m_nSendOut=0;          //则将该值设置为 0
                   }
        fSuccess=TRUE;                    //返回 TRUE 表示成功
 }
 LeaveCriticalSection(&m_csSend);         //离开临界区
 return fSuccess;                         //返回状态标志
}
void CSmsTraffic::PutRecvMessage(SM_PARAM* pparam, int nCount)
                                          //将短消息放入接收队列中
{
 EnterCriticalSection(&m_csRecv);         //进入临界区
 for (int i=0; i < nCount; i++)           //循环判断
 {
      memcpy(&m_SmRecv[m_nRecvIn], pparam, sizeof(SM_PARAM));
                                          //复制数据
      m_nRecvIn++;                        //接收消息队列长度值增加
      if (m_nRecvIn >= MAX_SM_RECV)       //如果接收消息队列长度大于规定值
            {
                  m_nRecvIn=0;            //设置接收消息队列长度值为 0
                  }
       pparam++;                          //增加计数值
 }
 LeaveCriticalSection(&m_csRecv);         //离开临界区
}
```

```
BOOL CSmsTraffic::GetRecvMessage(SM_PARAM* pparam)
                                                //从接收队列中取一条短消息
{
 BOOL fSuccess=FALSE;                           //定义并初始化变量
 EnterCriticalSection(&m_csRecv);               //进入临界区
 if (m_nRecvOut != m_nRecvIn)                   //判断消息队列长度是否相同
 {
        memcpy(pparam, &m_SmRecv[m_nRecvOut], sizeof(SM_PARAM));
                                                //复制数据
        m_nRecvOut++;                           //增加接收消息队列的长度值
        if (m_nRecvOut >= MAX_SM_RECV)          //如果队列长度值大于规定值
              {
                  m_nRecvOut=0;                 //设置队列长度值为 0
                  }
        fSuccess=TRUE;                          //设置状态标志位 TRUE
 }
 LeaveCriticalSection(&m_csRecv);               //离开临界区对象
 return fSuccess;
}
UINT CSmsTraffic::SmThread(LPVOID lParam)       //实现线程函数
{
 CSmsTraffic* p=(CSmsTraffic *)lParam;          //将参数强制转换为短消息类型
 int nMsg;                                      //收到短消息条数
 int nDelete;                                   //目前正在删除的短消息编号
 SM_BUFF buff;                                  //接收短消息列表的缓冲区
 SM_PARAM param[256];                           //发送/接收短消息缓冲区
 CTime tmOrg, tmNow;                            //上次和现在的时间，计算超时用
 enum {                                         //定义联合结构体
     stBeginRest,                               //开始休息/延时
     stContinueRest,                            //继续休息/延时
     stSendMessageRequest,                      //发送短消息
     stSendMessageResponse,                     //读取短消息列表到缓冲区
     stSendMessageWaitIdle,                     //发送不成功，等待 GSM 就绪
     stReadMessageRequest,                      //发送读取短消息列表的命令
     stReadMessageResponse,                     //读取短消息列表到缓冲区
     stDeleteMessageRequest,                    //删除短消息
     stDeleteMessageResponse,                   //删除短消息
     stDeleteMessageWaitIdle,                   //删除不成功，等待 GSM 就绪
     stExitThread                               //退出
 } nState;                                      //处理过程的状态
 nState=stBeginRest;                            //设置联合体
 while (nState != stExitThread)                 //发送和接收处理的状态循环
 {
       switch(nState)                           //比较状态
       {
             case stBeginRest:
                  TRACE("State=stBeginRest\n");
                  tmOrg=CTime::GetCurrentTime();
                                                //获取当前系统时间
                  nState=stContinueRest;        //为联合体赋值
                  break;                        //跳出循环
             case stContinueRest:
                  TRACE("State=stContinueRest\n");
                  Sleep(300);                   //线程函数暂停 0.3 秒
                  tmNow=CTime::GetCurrentTime();
```

```
                                                   //获取系统当前时间
            if (p->GetSendMessage(&param[0]))
                                                   //获取发送消息是否成功
            {
                nState=stSendMessageRequest;
                                                   //有待发短消息时，立即运行
            }
            else if (tmNow-tmOrg >= 5) //待发短消息队列空，休息 5 秒
            {
                nState=stReadMessageRequest;
                                                   //转到读取短消息状态
            }
            break;                                 //跳出循环
        case stSendMessageRequest:
             TRACE("State=stSendMessageRequest\n");
             gsmSendMessage(&param[0]);            //向短信猫发送消息
             memset(&buff, 0, sizeof(buff));       //初始化缓冲区
             tmOrg=CTime::GetCurrentTime();
                                                   //获取系统当前时间
             nState=stSendMessageResponse;
                                                   //为联合体赋值
             break;                                //跳出循环
        case stSendMessageResponse:
             TRACE("State=stSendMessageResponse\n");
             Sleep(100);                           //线程函数暂停 0.1 秒
             tmNow=CTime::GetCurrentTime();
                                                   //获取当前系统时间
             switch (gsmGetResponse(&buff))
                                                   //获取短信猫响应
             {
                   case GSM_OK:                    //查看短信猫响应码
                         TRACE("  GSM_OK %d\n", tmNow - tmOrg);
                         nState=stBeginRest;
                                                   //为联合体赋值
                         break;                    //跳出循环
                   case GSM_ERR:
                         TRACE("  GSM_ERR %d\n", tmNow - tmOrg);
                         nState=stSendMessageWaitIdle;
                         break;
                   default:                        //默认状态
                         TRACE("  GSM_WAIT %d\n", tmNow - tmOrg);
                         if (tmNow - tmOrg >= 10)
                                                   //查看是否超时大于 10 秒
                         {
                               TRACE("  Timeout!\n");
                               nState=stSendMessageWaitIdle;
                         }
             }
             break;                                //跳出循环
        case stSendMessageWaitIdle:
             Sleep(500);                           //线程暂停执行 0.5 秒
             nState=stSendMessageRequest;
                                                   //直到发送成功为止
             break;
        case stReadMessageRequest:
             TRACE("State=stReadMessageRequest\n");
             gsmReadMessageList();                 //读取信息列表
```

```
            memset(&buff, 0, sizeof(buff)); //初始化缓冲区
            tmOrg=CTime::GetCurrentTime();
                                             //获取系统当前时间
            nState=stReadMessageResponse;
            break;
        case stReadMessageResponse:
            TRACE("State=stReadMessageResponse\n");
            Sleep(100);
            tmNow=CTime::GetCurrentTime();
            switch (gsmGetResponse(&buff))
                                             //根据短信猫的响应码判断
            {
                case GSM_OK:                 //短信猫响应成功
                    TRACE("  GSM_OK %d\n", tmNow - tmOrg);
                    nMsg=gsmParseMessageList(param, &buff);
                    if (nMsg > 0)            //获取的消息数目大于 0
                    {
                        p->PutRecvMessage(param, nMsg);
                                      `      //获取接收到的消息
                        nDelete=0;           //初始化变量
                        nState=stDeleteMessageRequest;
                    }
                    else                     //若获取到的消息为空
                    {
                        nState=stBeginRest;
                    }
                    break;
                case GSM_ERR:                //短信猫响应错误
                    TRACE("  GSM ERR %d\n", tmNow - tmOrg);
                    nState=stBeginRest;
                    break;
                default:                     //短信猫其他响应码
                    TRACE("  GSM_WAIT %d\n", tmNow - tmOrg);
                    if (tmNow - tmOrg >= 15)
                                             //15 秒超时
                    {
                        TRACE("  Timeout!\n");
                        nState=stBeginRest;
                    }
            }
            break;
        case stDeleteMessageRequest:
            TRACE("State=stDeleteMessageRequest\n");
                                             //删除消息
            if (nDelete < nMsg)
            {
                gsmDeleteMessage(param[nDelete].index);
                                             //删除短信猫消息
                memset(&buff, 0, sizeof(buff));
                                             //初始化缓冲区
                tmOrg=CTime::GetCurrentTime();
                                             //获取系统当前时间
                nState=stDeleteMessageResponse;
            }
            else
            {
                nState=stBeginRest;
            }
            break;
```

```
            case stDeleteMessageResponse:
                 TRACE("State=stDeleteMessageResponse\n");
                 Sleep(100);
                 tmNow=CTime::GetCurrentTime();
                 switch (gsmGetResponse(&buff))
                                         //根据短信猫的响应码进行判断
                 {
                       case GSM_OK:
                            TRACE("  GSM_OK %d\n", tmNow - tmOrg);
                             nDelete++;
                             nState=stDeleteMessageRequest;
                             break;
                       case GSM_ERR:
                           TRACE("  GSM_ERR %d\n", tmNow - tmOrg);
                           nState=stDeleteMessageWaitIdle;
                           break;
                       default:
                            TRACE("  GSM_WAIT %d\n", tmNow - tmOrg);
                             if (tmNow - tmOrg >= 5)
                                         //5 秒超时
                             {
                                   TRACE("  Timeout!\n");
                                   nState=stBeginRest;
                             }
                }
                break;
            case stDeleteMessageWaitIdle:
                 TRACE("State=stDeleteMessageWaitIdle\n");
                 Sleep(500);
                 nState=stDeleteMessageRequest;
                                         //直到删除成功为止
                 break;
          }
          DWORD dwEvent=WaitForSingleObject(p->m_hKillThreadEvent, 20);
                                         //等待事件对象受信
          if (dwEvent == WAIT_OBJECT_0)  nState=stExitThread;
 }
 SetEvent(p->m_hThreadKilledEvent);     //将事件对象设置为受信状态
 return 9999;                           //返回数字
}
```

在上面的代码中，由于短消息类需要多线程的支持。所以，用户在该类的定义中添加了一个线程函数 SmThread()。当用户需要启动该线程函数时，调用函数 AfxBeginThread()启动即可。

实例程序中，用户为了避免使短消息类在同一时间对同一资源进行操作。所以，在程序中使用了事件对象以及临界区对象等实现线程同步的方法。

注意：用户在随书光盘中可以参考实例代码以及运行结果，以便分析在该类中，为什么需要使用多线程和线程同步等。

在本节中，主要向用户介绍了短消息类的基本组成和定义实现等，并且通过实例代码向用户讲解了短消息类的运行过程和成员函数的实现方法。在 14.6 节中，将向用户讲解本实

例程序的主要功能部分，即发送与接收。

14.6　发送和接收

用户从前面所学习到的知识中，已经对短信猫的相关硬件、AT 指令以及短消息类的封装等非常了解熟悉了。因此，在本节中，将主要向用户讲解实例界面的美化以及 PC 如何向短信猫发送或接收数据的相关知识。

14.6.1　创建实例工程界面

为了使实例程序拥有更好的交互性，用户应该使用 VC 的资源管理器对程序界面进行美化。本节将向用户介绍如何美化程序界面以及代码编写。

1．选择工程应用程序类型

用户创建本章实例工程时，使用 MFC 应用程序向导进行工程相关信息的设置。在设置步骤的第一步时，选择该工程的应用程序类型应该为“单文档”类型，如图 14.28 所示。

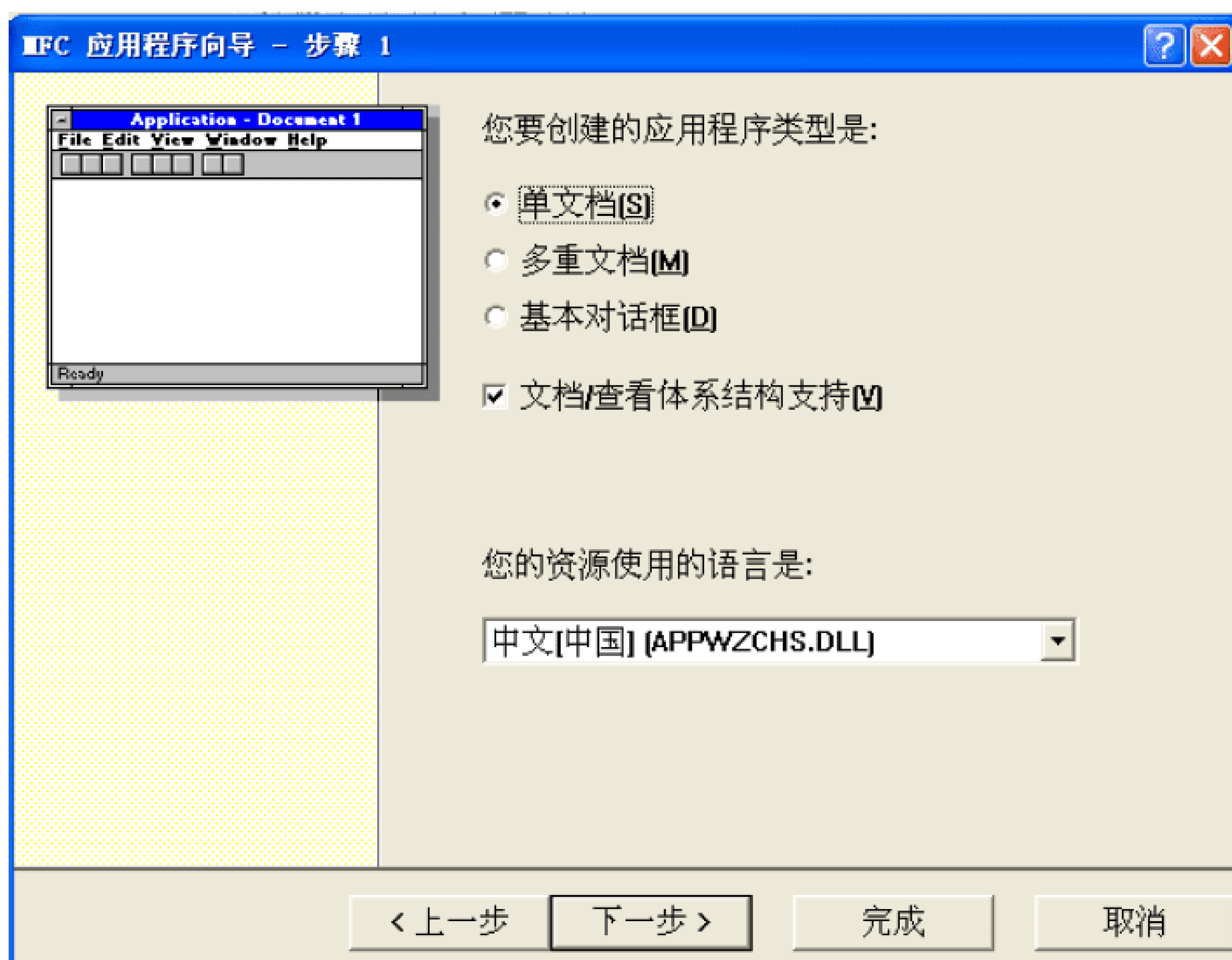


图 14.28　选择应用程序的类型为单文档

注意：用户在这一步进行设置时，也可以将应用程序类型指定为多文档类型。那么，用户便可以在实例工程中创建多个应用程序界面了。

2．添加参数设置对话框

在该实例工程中，用户需要为应用程序添加一个参数设置对话框，便于用户动态地输

入应用程序运行实现功能的相应信息。例如，与短信猫通信的串口号以及通信的波特率等。在 VC 资源管理器中，用户使用鼠标将所需控件等拖到对话框面板中，并设置其适合的大小，如图 14.29 所示。



图 14.29　参数设置对话框

注意：该对话框中，“短消息服务号码”是指用户的移动服务提供商的短消息中心号码。

参数设置对话框设计成功以后，用户可以在应用程序启动时，弹出该对话框。也可以在菜单中添加相应的菜单命令，弹出该对话框。如果用户选择在应用程序启动时弹出该对话框，则其代码如下：

```
BOOL CSmsTestApp::InitInstance()                       //应用程序初始化函数
{
    CSettingsDlg dlg;                                  //定义参数设置对话框类变量
    dlg.m_strPort=m_strPort;                           //为参数设置对话框中的各项赋值
    dlg.m_strRate=m_strRate;
    dlg.m_strSmsc=m_strSmsc;
    if (dlg.DoModal()==IDOK)                           //如果用户单击“确定”按钮
    {
            m_strPort=dlg_m_strPort;                   //则将用户输入的信息传给变量
            m_strRate=dlg_m_strRate;
            m_strSmsc=dlg_m_strSmsc;
    }
    else                                               //如果用户单击“放弃”按钮
    {
        return FALSE;                                  //则返回 false
    }
}
```

用户在程序中，添加以上代码，则在应用程序启动时，会首先弹出参数设置对话框。

如果用户选择从菜单命令中启动该对话框，则应该在程序框架的菜单项中，添加一个“设置”菜单项。首先，在 VC 的资源管理器中，添加一个新菜单项，并命名为“设置”。本实例中，该菜单项位于“文件”菜单下，如图 14.30 所示。

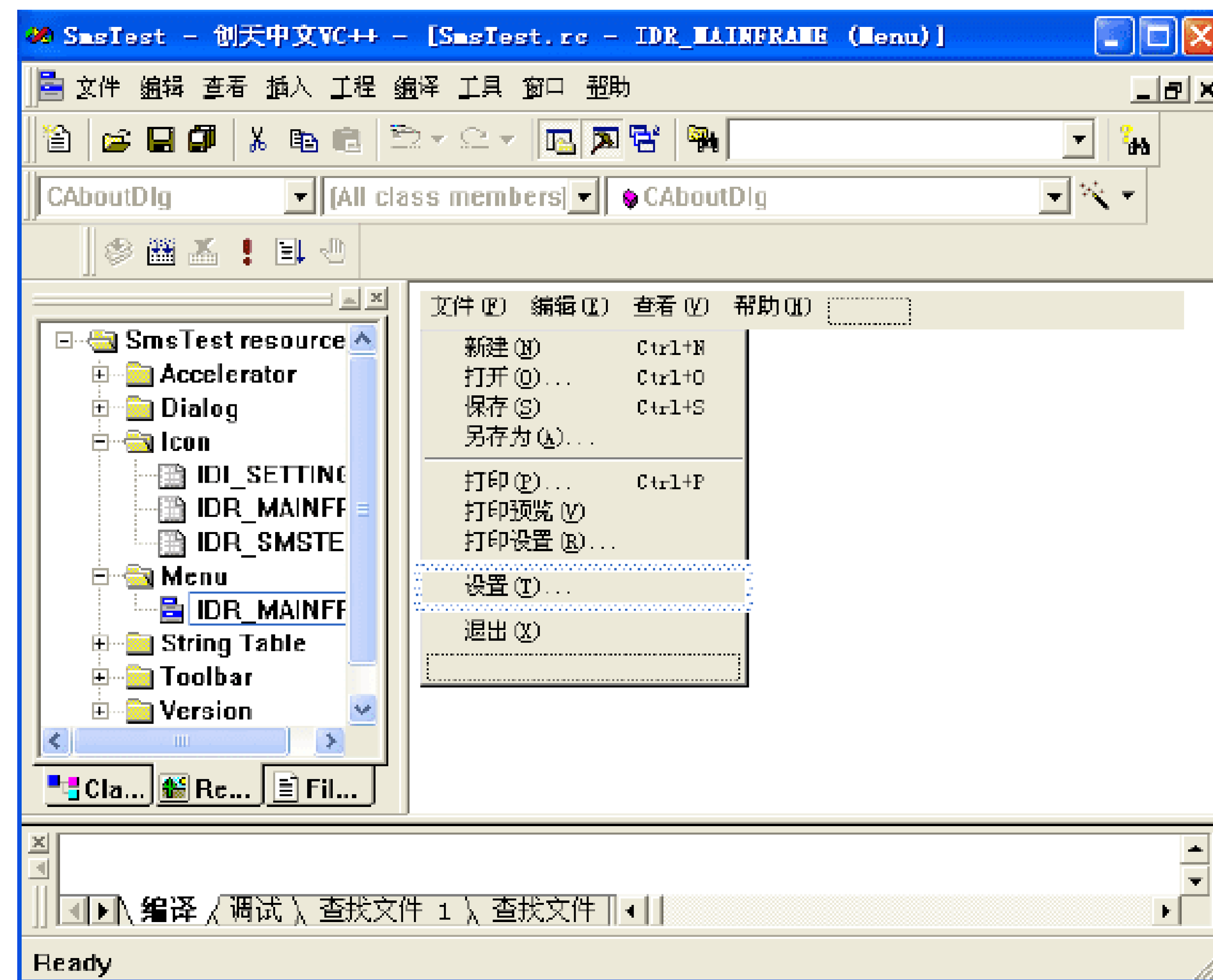


图 14.30　添加“设置”菜单项

然后，用户可以双击该菜单项，程序会弹出“菜单项目 属性”对话框。在该对话框中，用户可以为指定菜单项设置相应的菜单信息，如图 14.31 所示。

注意：在本章实例程序中，将该菜单项的 ID 修改为 ID_SETTING。这样，修改菜单项的 ID 方便于用户快速为其添加消息响应函数。

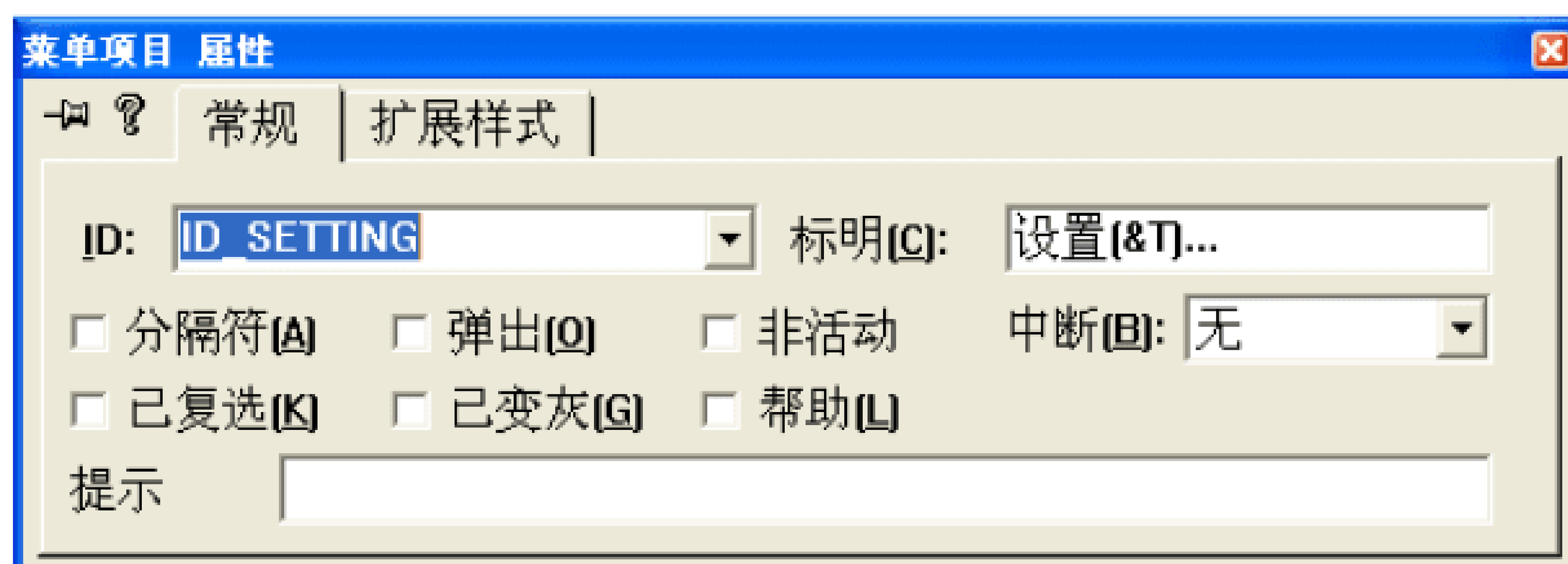


图 14.31　“菜单项目 属性”对话框

用户修改子菜单项的 ID 后，便可以为其添加菜单单击消息响应函数了。在 VC 主界面中，使用快捷键 Ctrl+W，打开 MFC ClassWizard 对话框，如图 14.32 所示。

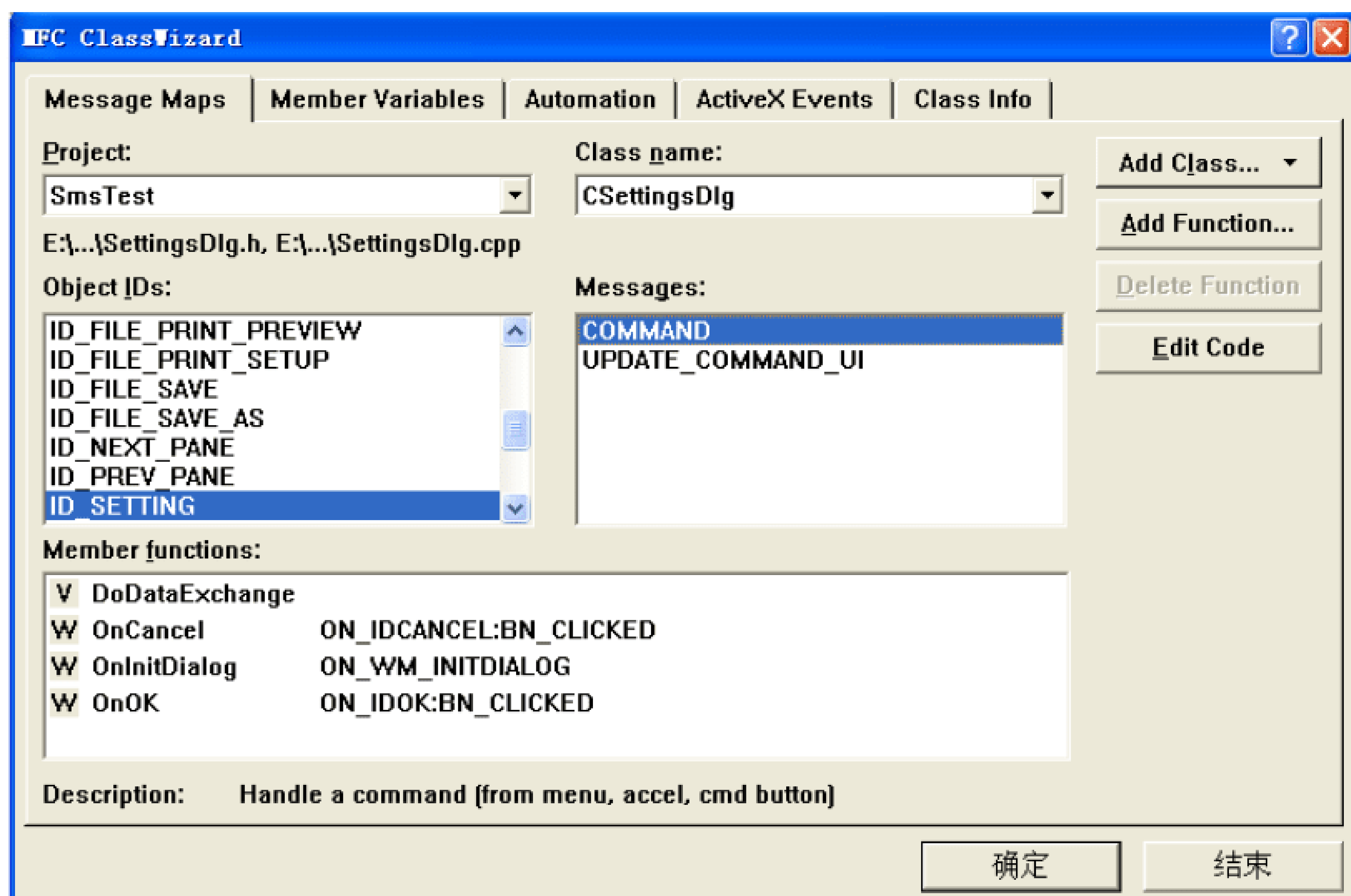


图 14.32　MFC ClassWizard 对话框

用户在 MFC 应用程序向导对话框中，先在 Object IDs 列表中，找到用户修改的对应菜单项 ID。再在 Messages 列表中选择 COMMAND。然后，单击 Add Function 按钮为该菜单项添加消息响应函数并可以修改响应函数名，如图 14.33 所示。

用户添加菜单消息响应函数成功后，可以将参数设置对话框的打开功能应用在该函数中，代码如下：

```
void CSmsTestApp::OnSetting()                    //菜单消息响应函数
{
 CSettingsDlg dlg;                               //定义参数设置对话框
 dlg.m_strPort=m_strPort;                        //初始化参数设置对话框中的参数
```

```
dlg.m_strRate=m_strRate;
dlg.m_strSmsc=m_strSmsc;
if(dlg.DoModal()==IDOK)                         //如果用户单击该对话框中的“确
                                                   定”按钮
{
        if(m_strPort !=dlg.m_strPort)           //判断新设置的端口是否相同
        {
             AfxMessageBox("端口设置在下次启动程序时生效");
                                                //显示消息框
        }
        m_strPort=dlg.m_strPort;                //保存各个变量值
        m_strRate=dlg.m_strRate;
        m_strSmsc=dlg.m_strSmsc;
}
}
```

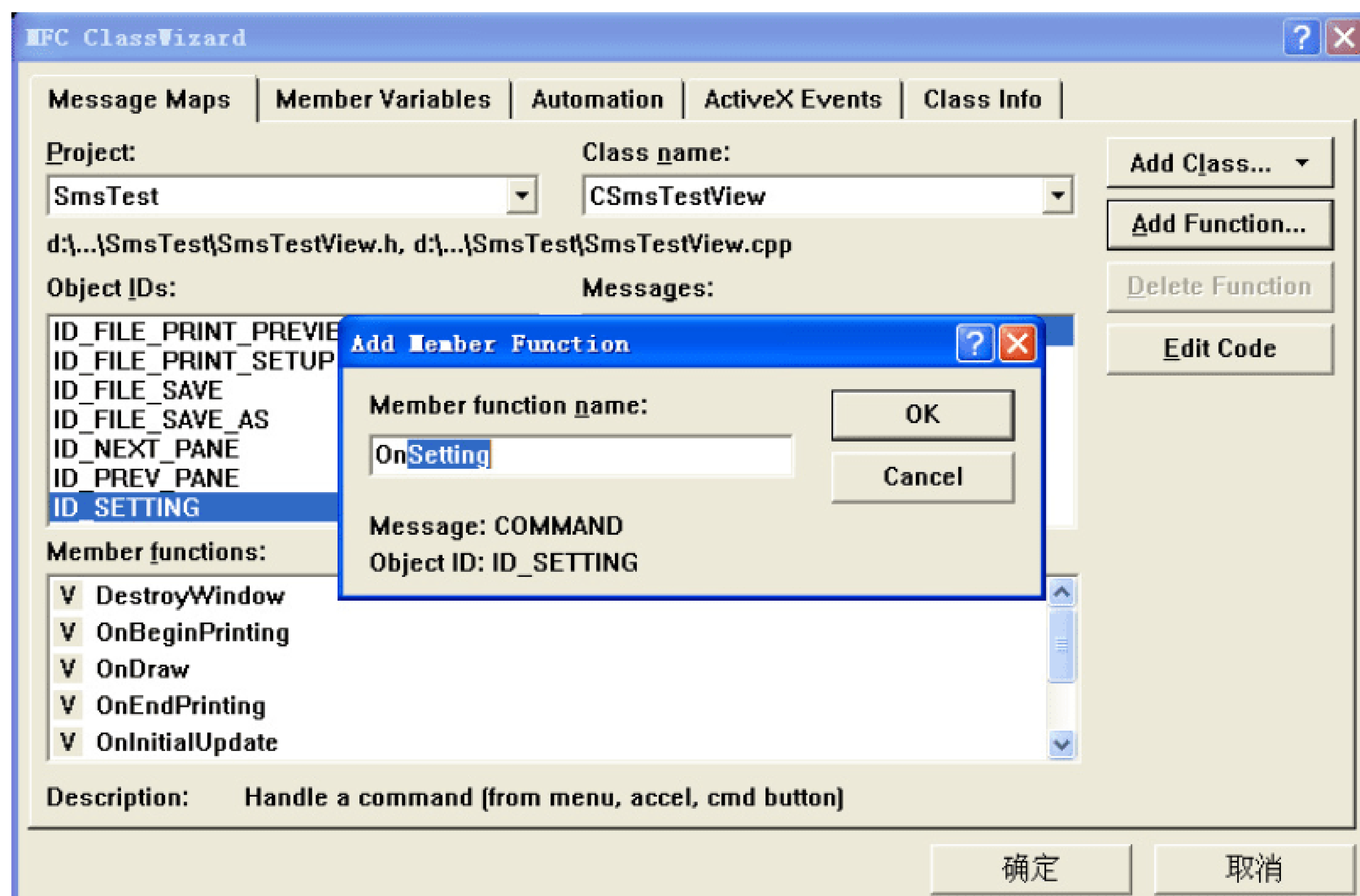


图 14.33　添加菜单消息响应函数并修改函数名

上面的代码主要是响应菜单项“设置”的消息响应函数。在该消息响应函数中，用户显示参数设置对话框以及初始化其中的参数值等。程序运行后，用户可以单击菜单项“设置”，程序会弹出参数设置对话框，如图 14.34 所示。

用户在使用过程中，如果需要临时修改串口号。那么，程序在修改完成之后，应该提醒用户此次修改将在下次启动程序时生效，如图 14.35 所示。



图 14.34　使用菜单项打开参数设置对话框

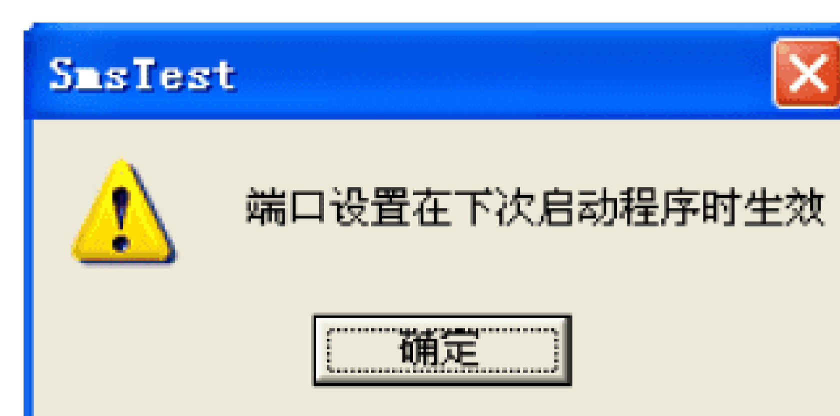


图 14.35　提示用户修改将在下次启动程序时生效

🔔注意：用户在实际编程时，本小节中所介绍的两种打开参数设置对话框的方法，最好仅选择其中的一种即可。

3．设置列表视图界面

为了应用程序界面的友好性和数据显示的直观性，用户还需要为应用程序界面添加列表视图界面等。

首先，用户在创建该工程时，必须在应用程序向导的最后一步，将视图类的基类修改为 CListView。这样，用户在程序中才能将界面进行列表显示，如图 14.36 所示。



图 14.36　修改工程视图类的基类

然后，在视图类的函数 PreCreateWindow()中为其添加列表报告模式。代码如下：

```
BOOL CSmsTestView::PreCreateWindow(CREATESTRUCT& cs)
{
 cs.style |= LVS_SHOWSELALWAYS | LVS_REPORT;            //添加列表报告样式
 return CListView::PreCreateWindow(cs);                 //返回其基类的该函数
}
```

🔔注意：函数 PreCreateWindow()是在程序创建窗口之前进行调用，所以，用户的许多初始化工作都可以在该函数中进行。关于结构体 CREATESTRUCT 的说明，请用户参考本书前面章节中相关的基础知识。

用户编译并运行以上程序，会发现在程序视图的最上方是一行灰色，如图 14.37 所示。



图 14.37　显示列表视图发生错误

出现以上错误的原因在于用户只是使视图窗口具有了列表视图的功能，但是并没有将其进行初始化。所以，用户会在视图窗口的最上方看见一行灰色。现在，用户在

视图类的初始化函数 OnInitialUpdate()中，进行列表的初始化。代码如下：

```
void CSmsTestView::OnInitialUpdate()                    //视图类初始化函数
{
 CListView::OnInitialUpdate();                          //调用基类的初始化函数
 CListCtrl& ListCtrl=GetListCtrl();                     //获取列表控件的指针对象
 ListCtrl.InsertColumn(0, "号码", LVCFMT_LEFT, 100);   //向列表中插入项目
 ListCtrl.InsertColumn(1, "时间", LVCFMT_LEFT, 140);
 ListCtrl.InsertColumn(2, "消息内容", LVCFMT_LEFT, 500);
 SetTimer(1, 1000, NULL);                               //启动定时器
}
```

在代码中，函数 GetListCtrl()是列表视图类中成员函数，其作用是用于获取并返回列表视图类对象的指针。该函数的原型如下：

```
CListCtrl& GetListCtrl( ) const;                    //获取并返回列表视图类的指针
```

该函数调用成功将返回 CListCtrl 类型的对象指针。然后，用户便可以使用该函数所返回的对象指针调用 CListCtrl 类的成员函数进行列表的初始化。其运行效果如图 14.38 所示。

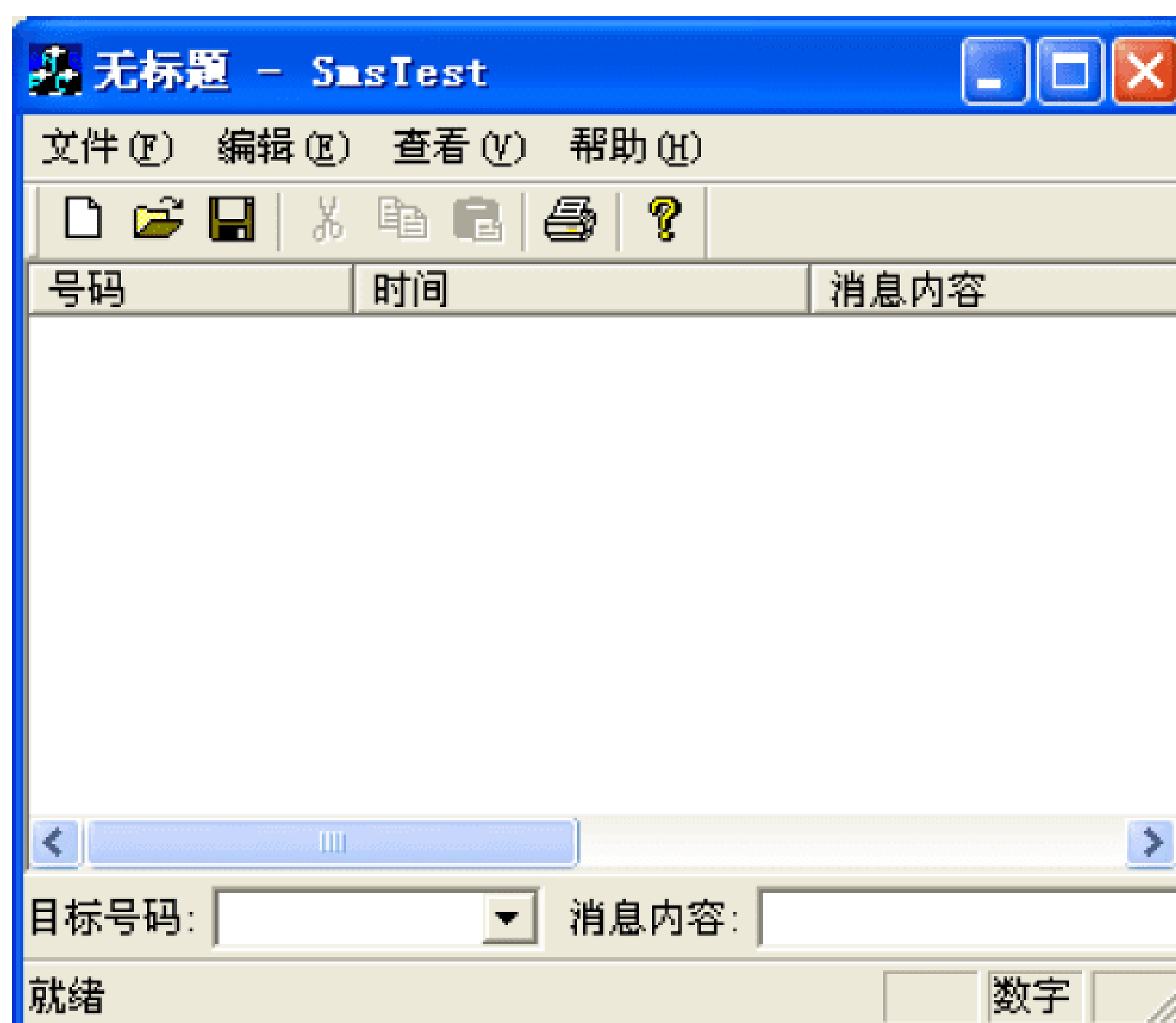


图 14.38　初始化列表视图

最后，用户调用函数 SetTimer()启动定时器，定时获取短信猫所返回的数据。在该工程实例中，用户将定时器的时间间隔设置为 1000 毫秒，表示程序将每间隔 1 秒调用一次定时器消息响应函数。代码如下：

```
void CSmsTestView::OnInitialUpdate()                    //视图类初始化函数
{
...                                                     //省略部分代码
 SetTimer(1, 1000, NULL);                               //启动定时器
}
```

定时器启动之后，用户需要在程序中定义定时器消息响应函数来响应定时器消息。实现该消息响应函数的定义必须在 VC 主界面中使用快捷键 Ctrl+W 打开应用程序向导对话框。在该对话框中，可以为工程视图类添加 WM_TIMER 消息，如图 14.39 所示。

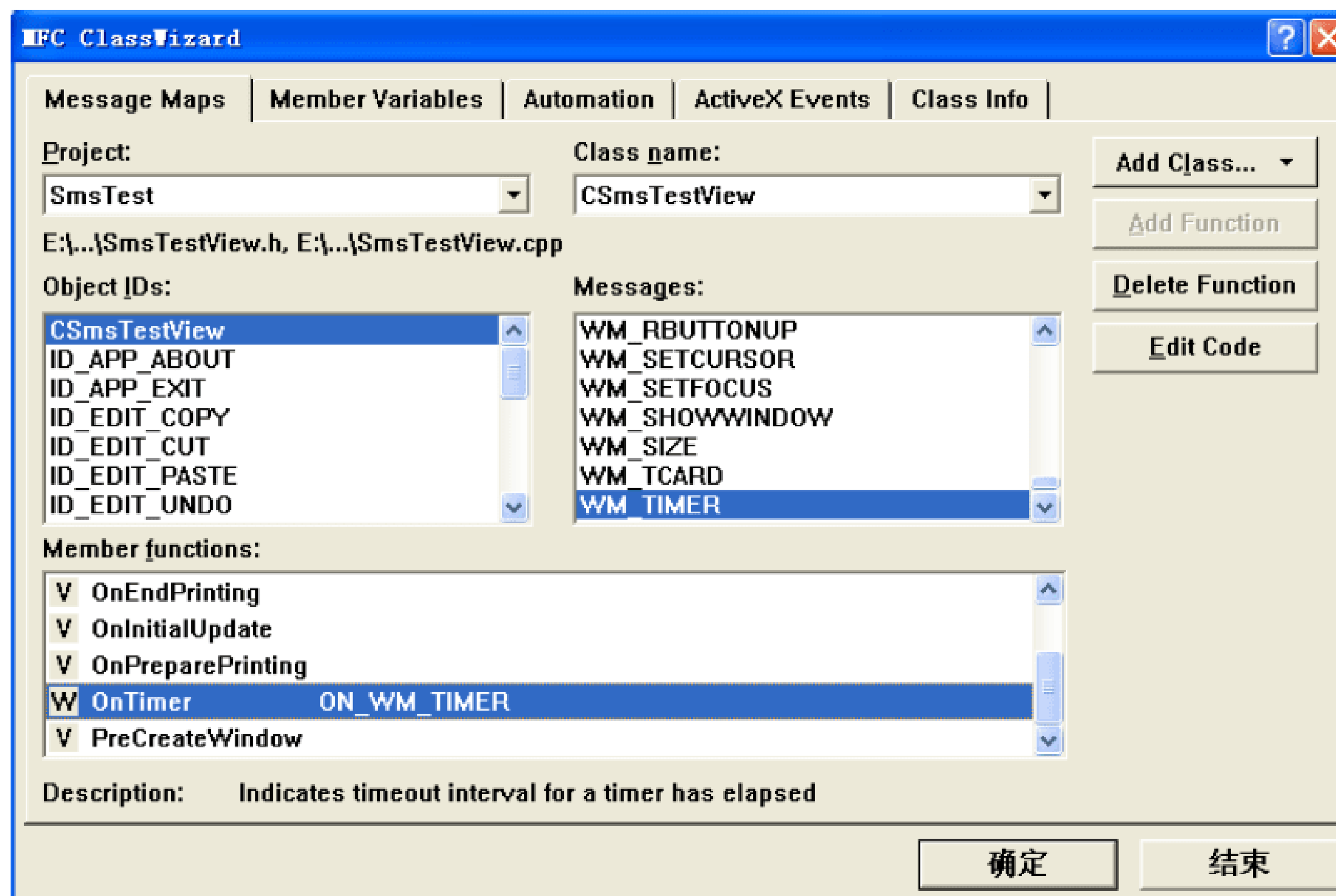


图 14.39　添加定时器消息响应函数

用户按照图 14.39 中所示的项目进行选择以后，单击 Add Function 按钮便可以成功为列表视图类添加定时器消息响应函数 OnTimer()。在该函数中，用户可以实现定时读取短信猫所返回的数据。代码如下：

```
void CSmsTestView::OnTimer(UINT nIDEvent)        //定时器消息响应函数
{
 if(nIDEvent == 1)                               //判断定时器 ID 是否为指定 ID
 {
      SM_PARAM SmParam;                          //结构体变量
      CString strTime;                           //定义字符串变量
      CString strNumber;
      CString strContent;
      CListCtrl& ListCtrl=GetListCtrl();
      if(theApp.m_pSmsTraffic->GetRecvMessage(&SmParam))
                                                 //获取接收到的短消息
      {
           strNumber=SmParam.TPA;                //获取短消息信息
           strContent=SmParam.TP_UD;             //获取短消息 ID
           strTime = "20" + CString(&SmParam.TP_SCTS[0],2)
                                                 //连接短消息基本信息字符串
               +"-"+CString(&SmParam.TP_SCTS[2],2)
               +"-"+CString(&SmParam.TP_SCTS[4],2)
               +" "+CString(&SmParam.TP_SCTS[6],2)
               +":"+CString(&SmParam.TP_SCTS[8],2)
               +":"+CString(&SmParam.TP_SCTS[10],2);
           if(strNumber.Left(2) =="86")          //去掉短消息号码前的 86
                 {
                      strNumber=strNumber.Mid(2);
                      }
           int nItemCount=ListCtrl.GetItemCount();//最多保留 200 条短消息
           if(nItemCount >=200)
           {
               ListCtrl.DeleteItem(0);           //删除短消息
```

```
                nItemCount--;                                    //列表索引自减
            }
            ListCtrl.InsertItem(nItemCount, strNumber); //插入新消息
            ListCtrl.SetItemText(nItemCount, 1, strTime);
            ListCtrl.SetItemText(nItemCount, 2, strContent);
            ListCtrl.EnsureVisible(nItemCount, FALSE);
        }
    }
    else
    {
        CListView::OnTimer(nIDEvent);           //调用基类的定时器响应函数
 }
}
```

注意：用户应该将接收短信猫返回数据的所有操作，均放在函数 OnTimer()中进行实现。这样，可以提高程序运行的效率。

在本节中，主要向用户讲解了列表视图的初始化以及定时读取短信猫所返回的数据等编程方法。如果用户在学习过程中，有不明白的地方请参考随书光盘中相应章节的实例代码进行参考。

4．添加发送功能对话框

在程序中，用户还应该使其具有发送 AT 指令等到短信猫执行的功能。因此，在工程中，用户首先需要添加一个发送功能的对话框并在该对话框中放入子控件等。在该工程实例中，将该对话框的名称修改为 IDD_SEND_SM，如图 14.40 所示。

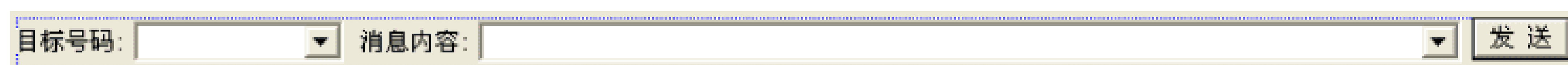


图 14.40　发送功能对话框

在程序启动时，将发送功能对话框显示在视图界面中。实现这一功能，用户需要将该对话框与 CDialogBar 类的对象相关联。用户在实例框架类 CMainFrame 中，定义 CDialogBar 类的对象 m_wndDialogBar。代码如下：

```
class CMainFrame : public CFrameWnd    //实例框架类
{
 CMainFrame();
 DECLARE_DYNCREATE(CMainFrame)
 protected:
 CStatusBar  m_wndStatusBar;           //状态栏对象
 CToolBar    m_wndToolBar;             //工具栏对象
 CDialogBar m_wndDialogBar;            //对话框条对象
}
```

然后，在框架类 CMainFrame 的创建函数 OnCreate()中，关联发送功能对话框和 CDialogBar 类对象。代码如下：

```
int CMainFrame::OnCreate(LPCREATESTRUCT lpCreateStruct)
                                               //框架类创建函数
{
 if (CFrameWnd::OnCreate(lpCreateStruct)==-1)  //判断基类创建函数执行情况
      return -1;
 if (!m_wndToolBar.CreateEx(this, TBSTYLE_FLAT, WS_CHILD | WS_VISIBLE |
```

```
CBRS_TOP|CBRS_GRIPPER|CBRS_TOOLTIPS|CBRS_FLYBY|CBRS_SIZE_DYNAMIC)||
      !m_wndToolBar.LoadToolBar(IDR_MAINFRAME))    //创建工具栏
 {
      TRACE0("Failed to create toolbar\n");
      return -1;
 }
 if (!m_wndStatusBar.Create(this) ||
      !m_wndStatusBar.SetIndicators(indicators,
        sizeof(indicators)/sizeof(UINT)))          //创建状态栏
 {
      TRACE0("Failed to create status bar\n");
      return -1;
 }
 if (!m_wndDialogBar.Create(this, IDD_SEND_SM,
      CBRS_BOTTOM|CBRS_TOOLTIPS|CBRS_FLYBY, IDD_SEND_SM))
                                               //创建对话框条并关联指定对话框
 {
       TRACE0("Failed to create dialog bar\n");
       return -1;
 }
   ...                                             //省略部分代码
 return 0;
}
```

在以上代码中，实现关联操作的代码是“m_wndDialogBar.Create(this, IDD_SEND_SM, CBRS_BOTTOM|CBRS_TOOLTIPS|CBRS_FLYBY, IDD_SEND_SM)”。当这段代码执行成功以后，用户将在程序运行效果中看到发送功能的对话框。如果代码执行失败，程序将在编译器输出窗口中输出信息 Failed to create dialog bar，如图 14.41 所示。

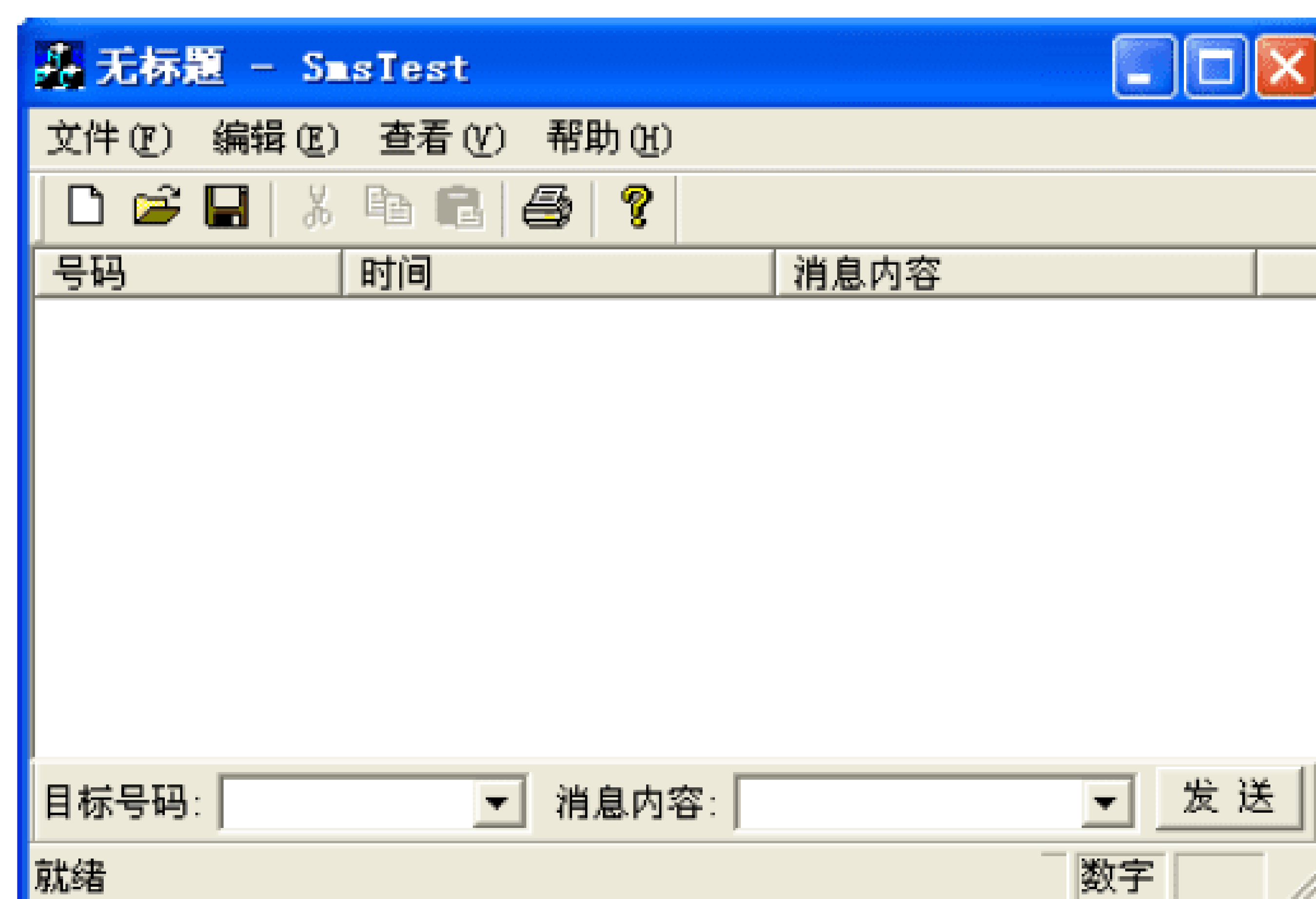


图 14.41　为程序添加发送功能对话框

由于本节中，仅向用户讲解如何进行程序界面设计步骤以及实现方法等。所以，关于发送功能的讲解将在 14.6.2 节中向用户进行讲解。

14.6.2　发送短信

在本章实例程序中，主要的发送功能都是在“发送”按钮的消息响应函数中进行实现。其实现原理是首先从控件中获取用户输入的信息。如果用户所输入的信息为空，则应该提示用户该信息不能为空。代码如下：

```
...                                                   //省略部分代码
 CComboBox* pNumberWnd=(CComboBox*)m_wndDialogBar.GetDlgItem(IDC_NUMBER)
;
 CComboBox* pContentWnd=(CComboBox*)m_wndDialogBar.GetDlgItem(IDC_CONTEN
T);
                                                      //获取编辑框的对象指针
CString strNumber;                                    //定义变量
 CString strContent;
 pNumberWnd->GetWindowText(strNumber);                //获取用户输入的号码
 pContentWnd->GetWindowText(strContent);              //获取用户输入的信息
 if(strNumber==NULL)
 {
         MessageBox("短消息中心服务号码或对方号码不能为空！");
                                                      //提示用户
         }
else
{
      if(strContent==NULL)
{
      MessageBox("短消息数据信息不能为空！");          //提示用户
      }
else
{
       ...                                            //进行相关数据的处理
       }
}
```

在代码中，用户为了安全的进行数据传输或接收，则应该首先检测用户输入的数据的正确性等。

然后，用户检测完数据的正确性以后，便可以对这些信息进行处理或发送到短信猫进行处理。代码如下：

```
void CMainFrame::OnSend()                                  //发送按钮消息响应函数
{
 CComboBox* pNumberWnd=(CComboBox*)m_wndDialogBar.GetDlgItem(IDC_NUMBER);
 CComboBox*
pContentWnd=(CComboBox*)m_wndDialogBar.GetDlgItem(IDC_CONTENT);
                                                           //获取编辑框的对象指针
 CString strSmsc;                                          //定义变量
 CString strNumber;
 CString strContent;
 strSmsc=theApp.m_strSmsc;                                 //获取短消息类的实例对象
 pNumberWnd->GetWindowText(strNumber);                     //获取用户输入的号码
 pContentWnd->GetWindowText(strContent);                   //获取用户输入的信息
 if(strNumber.GetLength() < 11)                            //检查号码的合法性
    {
         AfxMessageBox("请输入正确的号码！");                //提示用户重新输入号码
         pNumberWnd->SetFocus();                           //设置编辑框焦点
         pNumberWnd->SetEditSel(-1, 0);                    //设置组合框的当前位置
         return;
}
 CString strUnicode;                                   //检查短消息内容是否空或者超长
 WCHAR wchar[1024];
 int nCount = ::MultiByteToWideChar(CP_ACP, 0, strContent, -1, wchar, 1024);
 if(nCount <= 1)                                           //如果为空
```

```
{
     AfxMessageBox("请输入消息内容! ");              //提示用户
     pContentWnd->SetFocus();                      //设置编辑框焦点
     pContentWnd->SetEditSel(-1, 0);               //设置组合框当前位置
     return;                                       //返回空
}
else
   {
       if(nCount > 70)                             //将所有数据用 UCS2 编码，最大
                                                   //70 个字符（半角/全角）
{
   AfxMessageBox("消息内容太长，无法发送! ");       //提示用户
   pContentWnd->SetFocus();
   pContentWnd->SetEditSel(-1, 0);
   return ;
}
}
if(AfxMessageBox("确定发送吗? ",MB_YESNO)==IDYES)  //是否发送
{
      SM_PARAM SmParam;                            //结构体 SM_PARAM 变量
      memset(&SmParam, 0, sizeof(SM_PARAM));       //初始化缓冲区
      if(strSmsc[0] == '+')
        {
           strSmsc = strSmsc.Mid(1);
           }
      if(strNumber[0]=='+')
       {
           strNumber=strNumber.Mid(1);             //去掉号码前的+
           }
      if(strSmsc.Left(2) !="86")
       {
           strSmsc="86"+strSmsc;                   //在号码前加 86
           }
      if(strNumber.Left(2) !="86")
       {
           strNumber="86"+strNumber;
           }
      strcpy(SmParam.SCA, strSmsc);                //初始化短消息结构
      strcpy(SmParam.TPA, strNumber);
      strcpy(SmParam.TP_UD, strContent);
      SmParam.TP_PID=0;
      SmParam.TP_DCS=GSM_UCS2;
      theApp.m_pSmsTraffic->PutSendMessage(&SmParam); //发送短消息
      if(pNumberWnd->FindStringExact(-1,strNumber)<0)  //在列表中加入新字符
                                                       串
           {
               pNumberWnd->InsertString(0, strNumber);
               }
       if(pContentWnd->FindStringExact(-1,strContent)<0)
           {
               pContentWnd->InsertString(0, strContent);
               }
}
pContentWnd->SetFocus();                           //设置编辑框焦点
pContentWnd->SetEditSel(-1, 0);                    //设置组合框的当前位置
}
```

用户通过上面的代码，实现了程序通过制定端口发送指令等信息到短信猫，并由短信

猫执行相关指令将信息发送到目的号码手机中。首先，用户在目标号码中输入信息接收号码，在消息内容中，输入将要发送的短信内容。然后，单击“发送”按钮即可，如图 14.42 所示。

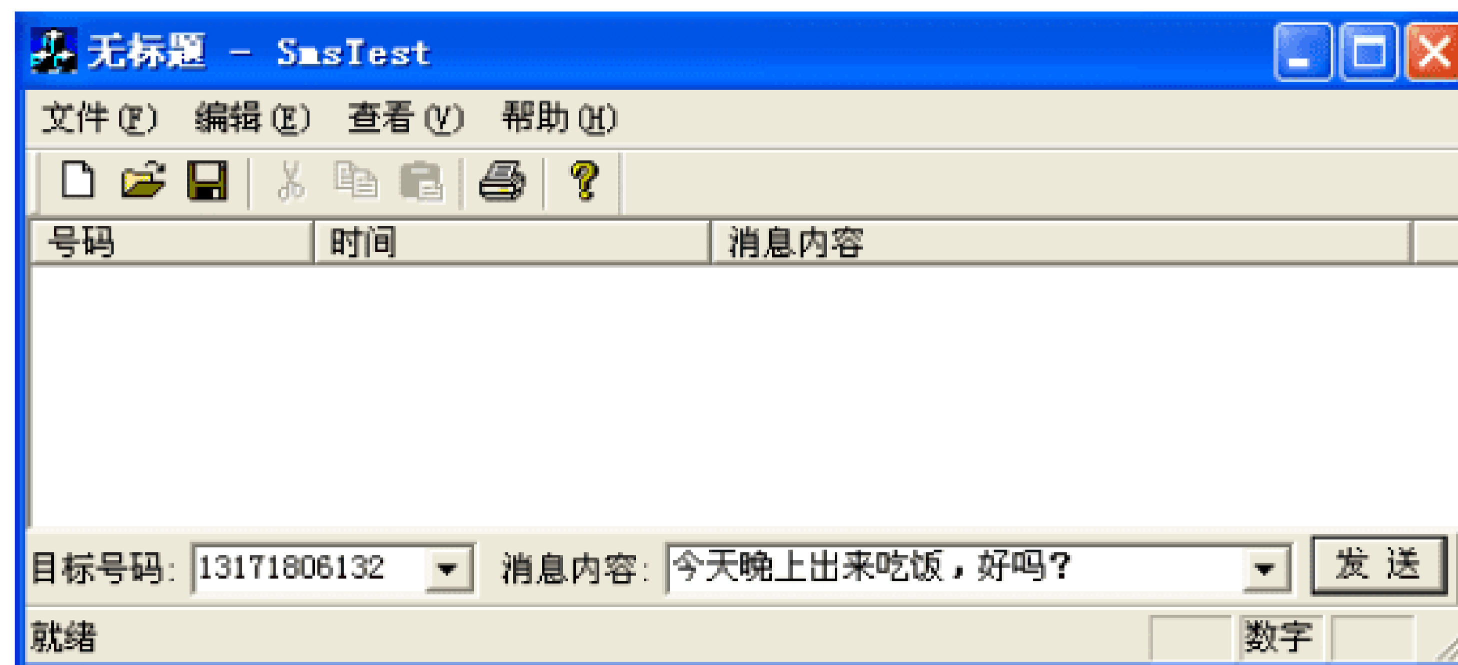


图 14.42　发送手机短消息

注意：*如果用户需要在发送指令或信息时，为程序添加其他功能，则可以参考随书光盘中的相应章节内容的实例代码进行添加或改写即可。*

14.6.3　接收短信

对于本章实例工程而言，实现短消息的接收功能也是非常重要的一部分。所以，本节将向用户介绍通过短信猫实现短消息数据接收的编程方法。

1．短消息接收步骤

首先，当短信猫接收到对方发送的短消息后，用户需要从短信猫中读取该短消息的数据。在前面的章节中，曾向用户介绍过短信猫的相关 AT 指令的用法。因此，在这里实现从短信猫中读取短消息时，用户可以使用 AT 指令 AT+CMGR 对短消息进行读取。在本章中，用户可以使用前面所封装的串口操作函数 ReadComm()进行短消息读取，代码如下：

```
 ...                                                     //省略部分代码
 CString str;                                            //定义命令字符串变量
 char buffer[1024]={0};                                  //定义数据缓冲区
 str="AT+CMGR";                                          //初始化命令字符串
 int i=WriteComm(str.GetBuffer(0),str.GetLength()); //通过串口向短信猫写入命
令字符串
 if(i!=0)
{
MessageBox("接收命令发送成功，正在接收短消息！");          //提示用户
ReadComm(buffer,1024);                              //通过串口向短信猫读取短信息
}
else
{
MessageBox("接收命令发送失败！请重试！");
}
...                                                      //省略部分代码
```

然后，用户再将读取到的短消息内容显示出来即可。代码如下：

```
...                                                    //省略部分代码
CString str1;                                          //定义字符串变量
str1.Format("%s",buffer);                              //格式化短消息
MessageBox(str1);                                      //显示消息框
...                                                    //省略部分代码
```

2. 短消息接收函数

在实例中，短消息的接收功能已经被封装在指定函数中。而由于实例程序必须定时查询短信猫中是否有新的短消息到来，所以，用户需要将短消息的接收功能在定时器消息 WM_TIMER 的消息响应函数中进行实现。代码如下：

```
void CSmsTestView::OnTimer(UINT nIDEvent)        //定时器消息响应函数
{
 if(nIDEvent ==2)                                 //判断定时器 ID 是否为指定 ID
 {
 CString str;                                     //定义命令字符串变量
 CString str1;                                    //定义字符串变量
 char buffer[1024]={0};                           //定义数据缓冲区
 str=”AT+CMGR”;                                   //初始化命令字符串
 int i=WriteComm(str.GetBuffer(0),str.GetLength());
                                              //通过串口向短信猫写入命令字符串
 if(i!=0)
 {
 MessageBox("接收命令发送成功，正在接收短消息！");   //提示用户
 ReadComm(buffer,1024);                           //通过串口向短信猫读取短消息
 }
 else
 {
MessageBox("接收命令发送失败！请重试！");
 }
 str1.Format("%s",buffer);                        //格式化短消息
 MessageBox(str1);                                //显示消息框
 }
 else
 {
    CListView::OnTimer(nIDEvent);                 //调用基类的定时器响应函数
 }
}
```

注意： 用户实际编程时，一定要注意启动定时器之前需要指定一个定时器的 ID，并且这个 ID 不能重复使用。例如，在上面的代码中，将定时器 ID 假设为 2，是因为在前面的代码中已经将 ID 为 1 的定时器使用了。所以，为了区别，这里将定时器 ID 设置为 2。

在本节中，主要讲解了创建实例工程与其界面的方法，并举例讲解了通过短信猫实现短消息发送以及接收的编程方法。在学习理论知识的同时，建议用户结合随书光盘中的实例代码进行，将达到更好的学习效果。

14.6.4　实现实例托盘程序

为了使实例程序更加人性化。所以，本章实例程序中加入了托盘程序的基本功能。本节将向用户主要讲解在 Windows 操作系统下，实现实例程序的托盘编程等相关知识。

1．托盘程序介绍

托盘程序是指当程序最小化或者程序线程挂起时，该程序将以图标的形式显示在系统界面的任务条右端。该位置也就是用户计算机上显示时间的那块区域，如图 14.43 所示。



图 14.43　系统托盘程序

用户在图 14.43 中，可以看见笔者机器上当前的所有托盘程序均以图标的方式进行显示的。当用户需要将该程序激活时，只需要使用鼠标双击相应的程序图标即可实现激活程序。

注意：Windows 系统下，托盘程序处于非活动状态。

2．托盘编程相关函数

在 Windows 编程中，实现程序托盘化的 API 函数是 Shell_NotifyIcon()，其函数原型如下：

```
WINSHELLAPI BOOL WINAPI Shell_NotifyIcon(
    DWORD dwMessage,
    PNOTIFYICONDATA pnid
);
```

如果该函数执行成功，则返回 true。否则，函数将返回 false。其参数意义分别如下：

- 参数 dwMessage 表示用户将要执行的操作类型。例如，添加、删除或修改图标。其取值如表 14.2 所示。

表 14.2　参数dwMessage取值

参数dwMessage取值	含 义 表 示
NIM_ADD	添加图标
NIM_DELETE	删除图标
NIM_MODIFY	修改图标

- 参数 pnid 是指向与托盘信息相关的结构体 NOTIFYICONDATA 变量。一般情况下，用户需要定义并初始化该结构体变量以后，才能成功调用该函数。结构体 NOTIFYICONDATA 定义如下：

```
typedef struct  NOTIFYICONDATA {
    DWORD cbSize;                              //该结构体长度
```

```
    HWND hWnd;                              //进行消息处理的窗口句柄
    UINT uID;                               //图标 ID
    UINT uFlags;                            //标识其哪些成员变量有效
    UINT uCallbackMessage;                  //自定义托盘消息
    HICON hIcon;                            //图标句柄
    char szTip[64];                         //程序托盘时显示的文字
} NOTIFYICONDATA, *PNOTIFYICONDATA;
```

例如，用户将程序实例实现托盘化。代码如下：

```
...                                           //省略部分代码
NOTIFYICONDATA nid;                           //定义结构体变量
nid.cbSize = (DWORD)sizeof(NOTIFYICONDATA);   //初始化结构体中的各个成员变量
  nid.hWnd = this->m_hWnd;
  nid.uID = IDR_MAINFRAME;
  nid.uFlags = NIF_ICON | NIF_MESSAGE | NIF_TIP ;
  nid.uCallbackMessage = WM_SHOWTASK;         //自定义的消息
  nid.hIcon=LoadIcon(AfxGetInstanceHandle(), MAKEINTRESOURCE(IDR_MAINF
RAME));
  strcpy(nid.szTip, "计划任务提醒");            //信息提示条为"计划任务提醒"
  Shell_NotifyIcon(NIM_ADD,&nid);             //在托盘区添加图标
  this->ShowWindow(SW_HIDE);                  //隐藏主窗口
```

在程序中，用户首先定义了结构体 NOTIFYICONDATA 变量 nid。然后，再对其中的成员变量进行初始化。最后，调用函数 Shell_NotifyIcon()实现托盘功能。其中，程序托盘时，当用户将鼠标移动到其图标位置时，会显示“计划任务提醒”，如图 14.44 所示。

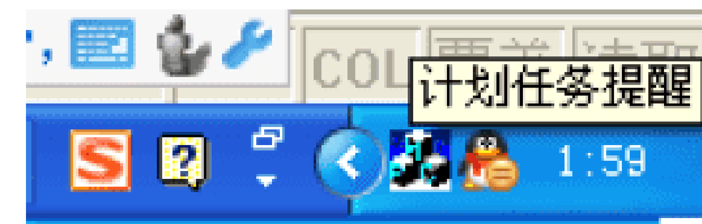


图 14.44　显示提示文本

注意：*在 Windows 系统中，用户实现托盘程序只需要使用函数 Shell_NotifyIcon()即可以实现。*

3. 程序实例托盘化

前面两节已经向用户讲解了实现托盘程序的相关方法。所以，在本节中将向用户讲解在本章实例程序中，如何编写添加托盘程序的相关代码等。

首先，用户需要在本章实例程序中定义一个自定义的消息，并将其名称修改为 WM_SHOWTASK。代码如下：

```
#define WM_SHOWTASK WM_USER+100                  //定义自定义消息
```

自定义消息定义成功后，在实例程序的框架类 CMainFrame 中定义消息 WM_SHOWTASK 的消息响应函数。代码如下：

```
 class CMainFrame : public CFrameWnd                  //框架类
{
protected:
 CMainFrame();
 DECLARE_DYNCREATE(CMainFrame)
 LRESULT onShowTask(WPARAM wParam,LPARAM lParam);    //声明消息响应函数
 }
...                                                   //省略部分代码
IMPLEMENT_DYNCREATE(CMainFrame, CFrameWnd)
BEGIN_MESSAGE_MAP(CMainFrame, CFrameWnd)
```

```
//{{AFX_MSG_MAP(CMainFrame)
ON_MESSAGE(WM_SHOWTASK,onShowTask)                    //添加消息映射
//}}AFX_MSG_MAP
END_MESSAGE_MAP()
...                                                   //省略部分代码
```

然后，在程序调用函数 Shell_NotifyIcon()实现托盘功能。由于在本章实例中，是将托盘功能实现在程序启动时。所以，用户在函数 CMainFrame::OnCreate()中需要添加代码如下：

```
int CMainFrame::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
if (CFrameWnd::OnCreate(lpCreateStruct) == -1)
      return -1;
NOTIFYICONDATA nid;                               //定义结构体变量
nid.cbSize = (DWORD)sizeof(NOTIFYICONDATA);      //初始化结构体中的各个成员变量
  nid.hWnd = this->m_hWnd;
  nid.uID = IDR_MAINFRAME;
  nid.uFlags = NIF_ICON | NIF_MESSAGE | NIF_TIP ;
  nid.uCallbackMessage = WM_SHOWTASK;              //自定义的消息
  nid.hIcon=LoadIcon(AfxGetInstanceHandle(),
MAKEINTRESOURCE(IDR_MAINFRAME));
     strcpy(nid.szTip, "计划任务提醒");            //信息提示条为计划任务提醒
     Shell_NotifyIcon(NIM_ADD,&nid);               //在托盘区添加图标
     this->ShowWindow(SW_HIDE);                    //隐藏主窗口
 return 0;
}
```

用户将上面的程序编译、运行后，会看到当实例程序启动时，该程序便已经实现了托盘化。并且在任务栏的右端添加了该实例程序的图标。那么，此时用户使用鼠标在该图标上进行操作，会发现实例程序并没有任何反应。这是因为用户在程序中，并没有为其相应的操作功能编写相应的代码。

最后，用户还要为自定义的托盘消息 WM_SHOWTASK 实现其消息响应函数。代码如下：

```
 LRESULT CMainFrame::onShowTask(WPARAM wParam,LPARAM lParam)
{
    {
        if(wParam==IDR_MAINFRAME)                   //wParam 接收的是图标的 ID
            return 1;
        switch(lParam)                              //lParam 接收的是鼠标的行为
        {
        case WM_RBUTTONUP:              //右键起来时弹出快捷菜单，这里只有一个关闭
            {
                LPPOINT lpoint = new tagPOINT;      //定义并初始化点结构
                ::GetCursorPos(lpoint);             //得到鼠标位置
                CMenu menu;                         //定义菜单对象
                menu.CreatePopupMenu();              //创建一个弹出式菜单
 menu.AppendMenu(MF_STRING, WM_DESTROY,"关闭"); //在弹出式菜单中添加关闭菜单
                menu.TrackPopupMenu(TPM_LEFTALIGN,  lpoint->x,  lpoint->y,
this);                                               //确定弹出式菜单的位置
                HMENU hmenu = menu.Detach();        //资源回收
                menu.DestroyMenu();                 //销毁菜单对象
                delete lpoint;                      //删除点结构体变量
```

```
            }
            break;
        case WM_LBUTTONDBLCLK:          //双击左键的处理
            {
                this->ShowWindow(SW_SHOW);
                                        //显示主窗口
            }
            break;
        }
        return 0;
    }
}
```

注意：上面的程序中，仅编写了部分基本功能的代码。如果用户需要为程序添加其他功能，则可以通过修改代码实现。

用户通过以上几个步骤，便完成了托盘程序的基本功能等代码的编写，其完整的功能代码请用户参考随书光盘中本小节的相关代码。所以，在这里将不再进行讲述。

14.7　相关代码分析

用户通过对本章前几个小节的学习，已经对使用 VC 编程实现通过短信猫发送手机短消息的方法有了进一步地了解和掌握。但是，用户可能对其中的某些编程方式或者方法并没有学习得非常透彻。因此，本节将专门向用户详细讲解本章中的一些关键代码段或编程思想。

14.7.1　参数设置对话框代码分析

在本章前面的相关章节中，已经向用户详细讲解了参数设置对话框的初始化等相关编程方法。但是，这些方法中并没有对该对话框中“确定”和“放弃”按钮的消息响应函数进行讲解。那么，在本节中将向用户补充讲解这部分的相关内容。

1．设置对话框界面优化

用户为了使程序界面具有较高的交互性，可以在参数设置对话框中添加一个图像控件。在本章实例中，将该控件的名称修改为 IDC_STATIC，并将其类型设置为“图标”，如图 14.45 所示。



图 14.45　修改图像控件的各个参数

如果控件参数修改完成之后，用户便可以选择将被现实的图标 ID 了。在实例程序中，为了向用户更好地讲解这部分知识。所以，在该实例中添加的图标资源为前面所使用的 QQ 图标，如图 14.45 所示，用户仅需要将已插入的图标资源 ID 修改为 IDI_SETTINGS 即可完成图标显示功能。最终显示图标的界面效果如图 14.46 所示。



图 14.46　显示图标后的对话框效果

注意：用户修改界面中的图标显示时，使用以上方法实现起来是非常简单可行的。

2. “确定”按钮响应函数

当用户将程序所需的参数设置好以后，单击确定按钮便可以使这些设置生效。因此，实现确定按钮的消息响应函数也非常重要。该函数的实现代码如下：

```
 void CSettingsDlg::OnOK()                           //确定按钮消息响应函数
{
 UpdateData();                                       //获取控件变量值
 int nSel;                                           //定义整型变量
 nSel=m_ctrlCommList.GetCurSel();                    //获取列表控件的选择项索引
 if(nSel>-1)                                         //若用户进行了选择
     {
      m_ctrlCommList.GetLBText(nSel,m_strPort);  //则获取用户选择索引处的项目
文字
      nSel=m_ctrlRateList.GetCurSel();               //获取用户选择的通信速率
if(nSel>-1)                                          //如果用户选择的项目索引正确
{
     m_ctrlRateList.GetLBText(nSel,m_strRate);  //获取通信速率值
 }
 else                                                //如果用户选择出错
 {
 if(m_strPort.IsEmpty()||m_strRate.IsEmpty()||m_strSmsc.IsEmpty())
                                                     //最后再判断用户输入的值是否为
空
 {
      AfxMessageBox("请正确设置端口和 SMSC!");       //弹出消息框显示错误信息
      return;                                        //返回空值
}
       }
 CDialog::OnOK();                                    //调用基类的相应函数
}
```

在代码中，变量 m_strRate 与 m_strPort 均为实例程序的全局变量，用户可以通过 MFC 应用程序向导为相应的控件添加变量。如果用户对这方面的内容不熟悉，请复习前面相应章节中的内容。

注意：当程序执行完这段代码以后，程序会获取到用户所选择或输入的相关参数，并将这些参数保存在相应的变量中。

3. “放弃”按钮响应函数

用户在该按钮的消息响应函数中，可以有两种功能。一种当用户单击该按钮后，程序自动退出。另外一种是当用户单击该按钮后，程序会将用户所输入的参数全部清空或恢复初始值。

如果用户在单击“放弃”按钮后，希望程序自动退出，则代码如下：

```
void CSettingsDlg::OnCancel()                     //放弃按钮消息响应函数
{
 ::SendMessage(GetParent()->m_hWnd,WM_CLOSE,0,0);  //向父窗口发送关闭消息
 CDialog::OnCancel();
}
```

用户可以试着编译以上代码，编译成功后并运行实例程序。但是当用户单击“放弃”按钮后，实例程序虽然退出了，但是系统却弹出了错误对话框，如图 14.47 所示。



图 14.47　系统弹出错误对话框

为什么程序在退出时，系统会弹出如图 14.47 所示的错误对话框？这是因为函数 SendMessage()的工作机制是基于同步模式的。也就是指当该函数将指定消息发送出去后，发送线程会等待其返回结果。在该实例中，由于发送线程是实例中的一个子窗口函数。所以，当函数 SendMessage()返回时，该窗口已经被迫退出并无法接收其返回的数据，因而造成错误。如果用户稍稍修改一下代码，可以避免该错误的发生。修改后的代码如下：

```
void CSettingsDlg::OnCancel()                     //放弃按钮消息响应函数
{
 ::PostMessage(GetParent()->m_hWnd,WM_CLOSE,0,0);  //向父窗口发送关闭消息
 CDialog::OnCancel();
}
```

用户可以编译并运行以上程序，按照前面相同的步骤进行操作，在关闭程序后，系统并没有弹出任何错误对话框。这是因为函数 PostMessage()使用的工作原理是基于异步模式的。也就是说当对话框通过该函数将关闭消息发送到实例程序的消息队列以后，会立即返回，避免了造成程序的不确定性。

注意： 用户在使用函数 PostMessage ()和 SendMessage()时，一定需要了解其工作原理。这样，用户在使用时，会使程序的运行效率提高很多。

14.7.2　发送功能代码分析

实例程序通过短信猫将短消息发送出去时，用户需要特别注意一些字符的判断或者转换。否则，短消息接收方所接收到的短消息可能出现无法显示等错误。首先，用户应该判断输入的号码或信息是否正确合法。代码如下：

```
        ...                                          //省略部分代码
if(strNumber.GetLength() < 11)                       //检查号码的合法性
    {
        AfxMessageBox("请输入正确的号码！");          //提示用户重新输入号码
        pNumberWnd->SetFocus();                      //设置编辑框焦点
        pNumberWnd->SetEditSel(-1, 0);               //设置组合框的当前位置
        return;
    }
    CString strUnicode;                              //检查短消息内容是否空或者超长
    WCHAR wchar[1024];
    int nCount = ::MultiByteToWideChar(CP_ACP, 0, strContent, -1, wchar,
    1024);
    if(nCount <= 1)                                  //如果为空
    {
        AfxMessageBox("请输入消息内容！");            //提示用户
        pContentWnd->SetFocus();                     //设置编辑框焦点
        pContentWnd->SetEditSel(-1, 0);              //设置组合框当前位置
        return;                                      //返回空
    }
    else
        {
            if(nCount > 70)                          //将所有数据用 UCS2 编码，最大
                                                     //70 个字符（半角/全角）
    {
        AfxMessageBox("消息内容太长，无法发送！"); //提示用户
        pContentWnd->SetFocus();
        pContentWnd->SetEditSel(-1, 0);
        return ;
    }
}
    if(AfxMessageBox("确定发送吗？",MB_YESNO)==IDYES)     //是否发送
    {
        SM_PARAM SmParam;                            //结构体 SM_PARAM 变量
        memset(&SmParam, 0, sizeof(SM_PARAM)); //初始化缓冲区
        if(strSmsc[0] == '+')
            {
                strSmsc = strSmsc.Mid(1);
                }
        if(strNumber[0]=='+')
            {
                strNumber=strNumber.Mid(1);          //去掉号码前的+
                }
        if(strSmsc.Left(2) !="86")
            {
                strSmsc="86"+strSmsc;                //在号码前加 86
                }
        if(strNumber.Left(2) !="86")
            {
```

```
                strNumber="86"+strNumber;
                }
        ...                                     //省略部分代码
```

在上面的代码中，用户首先判断了所输入的手机号码与短消息内容的正确性和合法性。如果这些信息为空或者非法，则程序会弹出错误提示对话框。

然后，用户通过对输入的信息进行验证后，便可以将短消息通过短信猫进行发送了。代码如下：

```
...                                             //省略部分代码
theApp.m_pSmsTraffic->PutSendMessage(&SmParam); //发送短消息
if(pNumberWnd->FindStringExact(-1,strNumber)<0) //在列表中加入新字符串
    {
        pNumberWnd->InsertString(0, strNumber);
        }
if(pContentWnd->FindStringExact(-1,strContent)<0)
    {
        pContentWnd->InsertString(0, strContent);
        }
}
    ...                                         //省略部分代码
```

在代码中，短消息通过封装的自定义函数 PutSendMessage()发送出去，并且将发送出去的短消息内容再显示到本地程序的列表中。

注意：函数 PutSendMessage()的相关定义以及作用，请用户参考 14.5.3 节中的相关内容。

14.7.3　接收功能代码分析

用户接收短消息是通过定时读取程序串口的数据缓冲区获取的。在接收短消息时，最重要的一步是判断数据的完整性和正确性。首先，用户应该获取短消息的相关信息。例如，短消息发送方的号码等。代码如下：

```
    ...                                         //省略部分代码
if(theApp.m_pSmsTraffic->GetRecvMessage(&SmParam))
                                                //获取接收到的短消息
        {
            strNumber=SmParam.TPA;              //获取短消息信息
            strContent=SmParam.TP_UD;           //获取短消息 ID
            strTime = "20" + CString(&SmParam.TP_SCTS[0],2)
                                                //连接短消息基本信息字符
                                                  串
                +"-"+CString(&SmParam.TP_SCTS[2],2)
                +"-"+CString(&SmParam.TP_SCTS[4],2)
                +" "+CString(&SmParam.TP_SCTS[6],2)
                +":"+CString(&SmParam.TP_SCTS[8],2)
                +":"+CString(&SmParam.TP_SCTS[10],2);
    ...                                         //省略部分代码
```

通过上面的代码，用户获取了短消息发送方的号码及其短消息内容。接下来，用户就需要对这些接收到的数据进行重组，使其结构到达最简。代码如下：

```
...                                            //省略部分代码
if(strNumber.Left(2) =="86")                   //去掉短消息号码前的 86
            {
                strNumber=strNumber.Mid(2);
                }
        int nItemCount=ListCtrl.GetItemCount();//最多保留 200 条短消息
    if(nItemCount >=200)
    {
        ListCtrl.DeleteItem(0);                //删除短消息
        nItemCount--;                          //列表索引自减
    }
    ListCtrl.InsertItem(nItemCount, strNumber); //插入新消息
        ListCtrl.SetItemText(nItemCount, 1, strTime);
        ListCtrl.SetItemText(nItemCount, 2, strContent);
        ListCtrl.EnsureVisible(nItemCount, FALSE);
    }
```

用户通过上面的程序段，不仅将短消息中的多余信息删除了，而且还将精简后的短信显示在程序界面的列表控件中。

注意：请用户参考随书光盘中的实例代码进行学习，并且可以对光盘中的实例代码进行修改达到学习的最佳目的。

本节主要向用户详细讲解了实例中一些重要的代码或步骤。希望用户通过本章的学习能够掌握将计算机与其他硬件设备进行结合编程的基本方法。同时，用户也可以在学习的过程中，通过其他专业书籍对短信猫的硬件结构或 AT 指令进行更为详细的学习。这样，对于本章实例的学习会起到很好的帮助。

14.8　小　　结

通过本章的学习，用户对于使用短信猫与 PC 相结合实现短消息发送功能的原理与功能代码编写的相关知识应该有一个非常清晰的认识。用户可以在本章实例代码中，进行自定义功能的代码编写等。这样，用户将对本章的相关知识点可以达到很好的学习效果。实例代码请用户参考随书光盘中的相应小节。